



李维 作品系列



Delphi 7

高效数据库程序设计

李维 著

Delphi 6/Kylix 1/2/3 适用

Borland新一代dbExpress提供了跨平台的高效率的数据库引擎,dbExpress不但提供广泛的关系型数据库存取能力,并且适用于客户机/服务器、Web和多层的应用系统。本书不但深入讨论dbExpress的使用技巧,更辅以丰富的范例让读者完全掌握 Delphi/Kylix 的 dbExpress 和 DataSnap 技术。



附 赠



机械工业出版社
China Machine Press



第一部分

dbExpress基本功能篇



第1章 dbExpress组件、概念、技术和应用程序

Delphi 6/7的重要功能之一便是推出了新一代跨平台的数据访问引擎——dbExpress。dbExpress是一组新的组件、技术和驱动程序，以允许程序员使用它连接到各种数据源，再配合不同的数据库连接 DLL文件，让程序员可以处理后端数据库中的数据。由于 dbExpress具备了跨平台的能力，能够同时在 Windows和Linux平台以及未来的 .NET上使用，更提供了快速的数据处理能力，让程序员能够开发出更有效率的数据库应用程序，因此势必成为以后 C++Builder/Delphi和Kylix的核心数据访问技术。

这些新的dbExpress组件除了新进入 Delphi世界的程序员需要学习之外，即使是使用 Delphi已经有一段时间的有经验的 Delphi程序员也需要上手的时间。此外，dbExpress提供了强大的功能，允许 Delphi程序员微调它的执行行为，并且访问低层的核心信息。同时dbExpress也能够与 DataSnap技术（这是 Delphi 6/7中的名称，在旧的 Delphi版本中称为 MIDAS）结合以便让程序员能够同时开发单机、Briefcase、主从结构和瘦客户类型的数据库应用程序，让程序员能够使用一组组件和技术同时开发数种类型的应用系统。

本书将会讨论 dbExpress的所有强大功能，不过万丈高楼平地起，本章将从说明如何使用这些新的 dbExpress组件开始，一步一步地带领各位学习到 dbExpress最精髓的核心技术。

1.1 dbExpress组件

在 Delphi 7 中，dbExpress组件组包含了7个组件，这些组件的功能就是让应用程序连接后端数据库，访问数据表中的数据，把修改的数据更新回数据库中以及让程序员观察 dbExpress向后端数据库下达的命令等。简单地说，这些组件涵盖了数据库应用程序所有必要的功能。图 1-1 是 dbExpress组件面板中的所有 dbExpress组件。

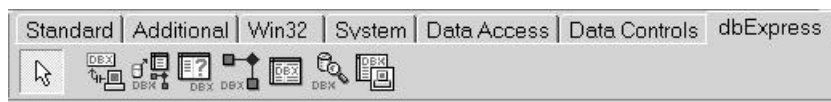


图1-1 Delphi 7的dbExpress组件组

下面的表格概要地说明了每一个 dbExpress组件的基本功能，在稍后的章节中将会详细讨论如何使用每一个 dbExpress组件。

组件名称	功 能
TSQLConnection	与后端数据库建立连接的组件
TSQLDataSet	可以用来执行SQL语句、执行存储过程或是直接连接数据库中特定数据表的组件。它算是一个通用的组件。TSQLDataSet同时具备了类似于TSQLQuery、TSQLStoredProc和TSQLTable组件的能力
TSQLQuery	用来执行SQL语句的组件
TSQLStoredProc	用来执行数据库中的存储过程（Stored Procedure）的组件
TSQLTable	用来连接数据库中特定的数据表的组件，类似于BDE中的TTable组件
TSQLMonitor	可以观看和检视客户端向后端数据源发出的SQL语句的组件。程序员可以使用它来调试或是调整应用程序的性能
TSimpleDataSet	允许dbExpress修改数据的组件，可以结合Delphi的数据感知组件以访问数据

在Delphi 7.0和Kylix 3.0中，dbExpress目前正式支持6种数据库，这些支持的数据库以及对应的dbExpress驱动程序总结在下面的表格中。

数据库名称	dbExpress驱动程序
InterBase 6.5/6.x	DBEXPINT.DLL
DB2 7.2	DBEXPDB2.DLL
Oracle 9i	DBEXPORA.DLL
MySQL 3.23.49	DBEXPMYS.DLL
MS SQL Server 2000	DBEXPMSS.DLL
Informix SE	DBEXPINF.DLL

虽然目前dbExpress只支持6种数据库，但是Borland已经声明将继续提供其他数据库的dbExpress驱动程序，例如Sybase System 12等。在新的dbExpress驱动程序开发完毕之后，Borland会公布在网站上让程序员下载，并且包含在随后版本的Delphi和Kylix中。

学习dbExpress组件最好的方法就是直接使用它来开发数据库应用程序，在下一个小节中，我们将立刻开始学习dbExpress组件组，让程序员快速地学习如何使用这些新的组件开发应用系统。

1.2 建立第一个dbExpress数据库应用程序

现在就让我们快速地使用dbExpress来开发一个数据库应用程序，学习如何使用dbExpress组件来访问数据。要连接数据库并且从其中访问数据，程序员可以使用下面的三个步骤来完成这个工作：

- 1) 使用TSQLConnection组件连接数据库。
- 2) 使用TSQLDataSet组件访问数据。
- 3) 在数据感知组件中显示数据。

现在就让我们一步一步地完成上面的三个步骤。

步骤1: 使用TSQLConnection组件连接数据库

首先点击Delphi的File|New|Application菜单建立一个新的Delphi应用程序，接着点击组件面板中的dbExpress选项卡，选择第一个组件TSQLConnection并且将它放入到应用程序的主窗体，如图1-2所示。有了TSQLConnection组件之后，现在我们需要让它连接到数据库服务器，在这个范例中是使用InterBase，读者可以使用其他数据库，例如Oracle或MySQL等。

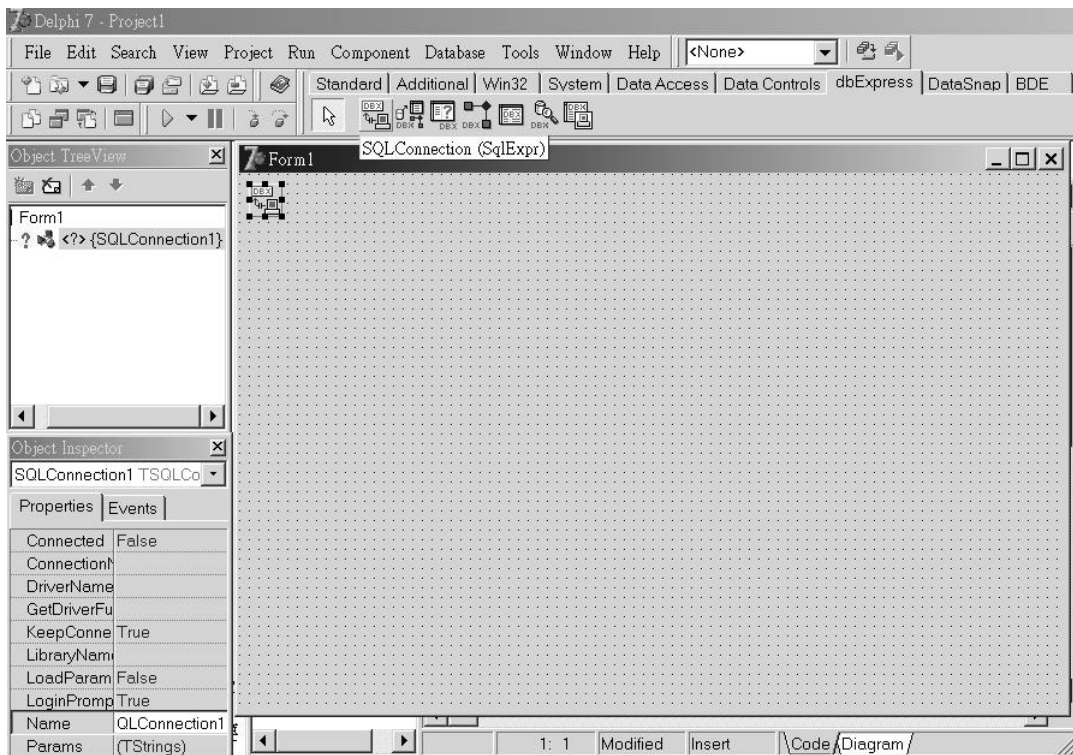


图1-2 在主窗体中放入TSQLConnection组件以连接数据库

要使用TSQLConnection连接数据库，请使用鼠标双击TSQLConnection，此时会出现TSQLConnection的组件编辑器，如图1-3所示。在图1-3中列出了目前dbExpress内置的连接或是用户新增的连接，由于现在本范例要使用InterBase作为连接的数据库，因此请使用鼠标点击上方的“+”按钮以建立一个新的连接。

此时Delphi会显示图1-4所示的新连接对话框，请在这个对话框中选择使用InterBase驱动程序，并且输入一个连接名称。在这个范例中使用了CHINESEDEMO作为本范例的连接名称。

接着Delphi会显示图1-5所示的对话框，要求输入CHINESEDEMO真正的数据库

路径和名称信息，请按照图 1-5 那样输入数据库的实际位置。由于 CHINESEDEMO 使用的 InterBase 数据库是使用 BIG5 内码建立的，因此请读者记得将 ServerCharSet 的特性值改为 BIG5。在输入了数据库实际位置之后，读者可以点击对话框上方的“~”按钮，以测试是否可以正确地连接到数据库。最后确定一切正确之后，点击 OK 按钮以完成设置 TSQLConnection 组件的步骤。

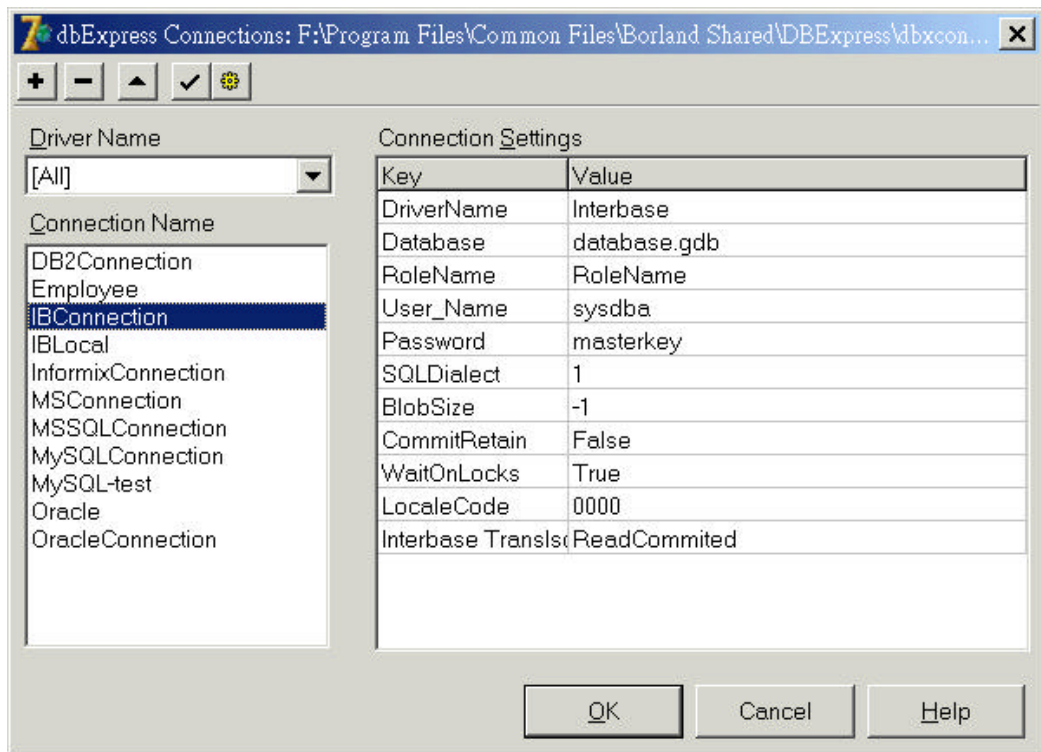


图 1-3 TSQLConnection 组件的组件编辑器

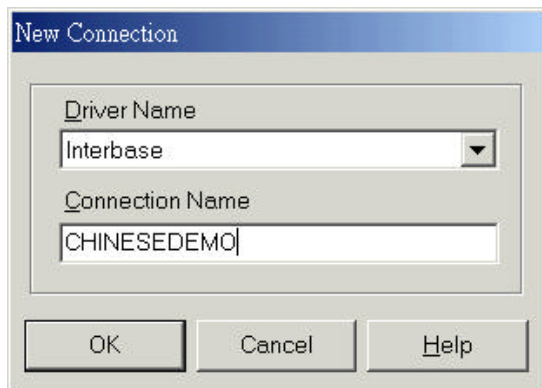


图 1-4 dbExpress 的新数据库连接对话框

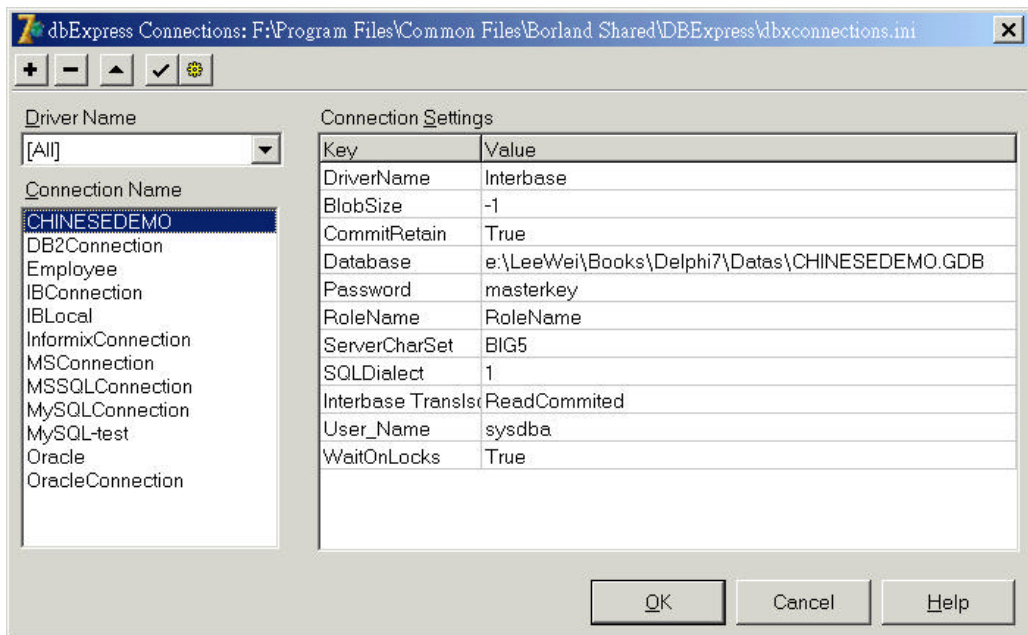


图1-5 输入CHINESEDEMO要连接的InterBase数据库

读者可以在本书的光盘中找到 CHINESEDEMO.GDB 这个 InterBase 数据库。

如果读者仔细观察图 1-5 所示的对话框，便会发现刚才设置的 CHINESEDEMO 这个 InterBase 连接信息 事实上是 存储在 \Borland Shared\DBExpress 目录下的 dbxconnections.ini 文件中，这是一个文本文件，读者也可以使用文字编辑器，例如 NotePad 或是 Delphi 的编辑器来修改其中的内容。

现在请点击对象检视器，设置 TSQLConnection 的 LoginPrompt 特性值为 False，以避免出现登录对话框，最后再把 Connected 特性值设置为 True。如此一来我们就成功地连接到 InterBase 服务器了（当然，读者的 InterBase 必须正在执行）。

步骤2：使用 TSQLDataSet 组件访问数据

现在再从 dbExpress 选项卡中选择第二个组件 TSQLDataSet，并且将它放到主窗体中。先在对象检视器中将它的 SQLConnection 特性值设置为步骤 1 放入的 SQLConnection1，再点击它的 CommandText 特性值。此时 Delphi 便会显示 CommandText 特性值的特性值编辑器，让程序员可以使用可视化的方式下达 SQL 命令。图 1-6 便是激活 CommandText 特性值编辑器的画面。

当 CommandText 特性值编辑器出现时，它会自动地从连接的数据库中取得所有目前可以被看到的数据表名称，放入到左上方的 Tables 列表框中，而把选择的数据表

的字段信息呈现在左下方的列表框中，真正的 SQL 命令则在右边的 Memo 控件中。至于什么数据表会出现在左上方的 Tables 列表框中则是由 TSQLConnection 的 TableScope 特性值来决定的。

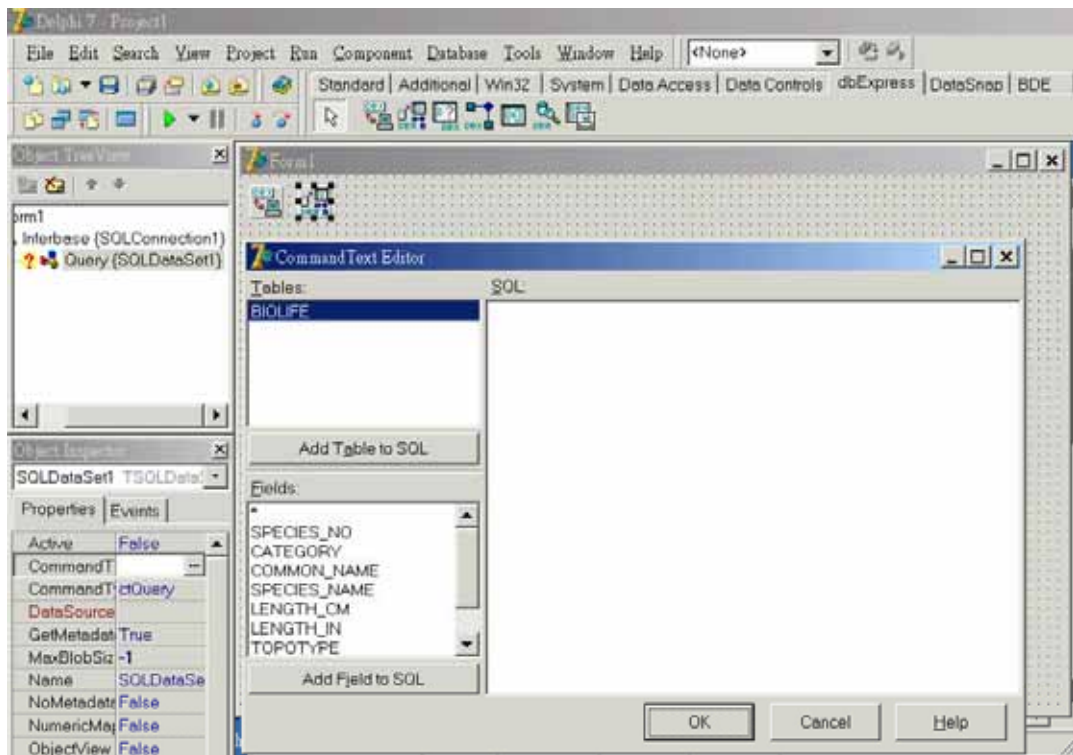


图1-6 TSQLDataSet的CommandText特性值编辑器

由于CHINESEDEMO中只有一个数据表 BIOLIFE，因此请使用鼠标点击左上方的BIOLIFE，再点击左下方的“*”以代表要从BIOLIFE数据表中取得所有字段的数据，如此一来CommandText特性值编辑器便会在右边的 SQL 窗口中自动产生 Select * from BIOLIFE的SQL命令，如图1-7所示。

请点击OK按钮，Delphi便会把刚才完成的 SQL 命令存储在 TSQLDataSet 组件的 CommandText 特性值中。接着请在主窗体中放入一个 TDataSource 组件，将它的 DataSet 特性值设置为刚才的 TSQLDataSet 组件，再放入一个 TDBNavigator 组件，将它的 DataSource 特性值设置为刚放入的 TDataSource 组件。

现在我们就可以通过把 TSQLDataSet 的 Active 特性值设置为 True 将数据从 BIOLIFE 数据表取到应用程序中。接着让我们激活 TSQLDataSet 的字段编辑器，把代表 BIOLIFE 每一个字段的字段对象加入应用程序中。把字段对象加入应用程序中的好处是程序员可以使用 Delphi 的对象检视器来设置字段对象的特性，例如字段显示的中文名称，或是设置字段显示的栏宽等。

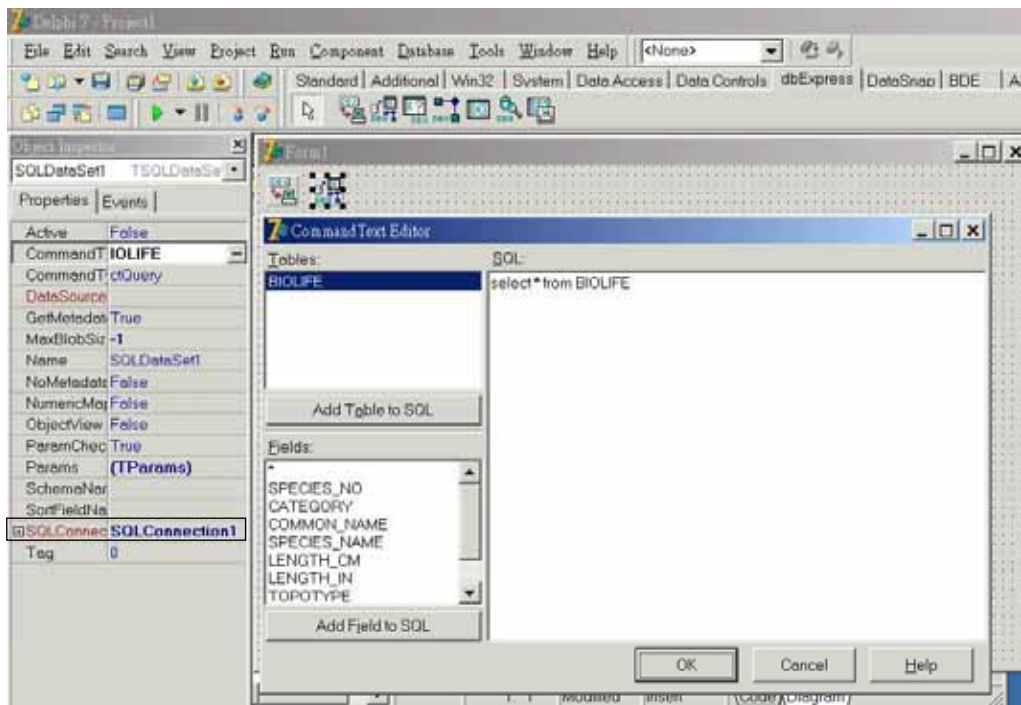


图1-7 使用CommandText特性值编辑器完成SQL命令

请点击主窗体中的 TSQLDataSet 组件，按下鼠标右键，Delphi 会显示一个快捷菜单，菜单中的第一个选项 Fields Editor... 便可以激活字段编辑器，如图 1-8 所示。

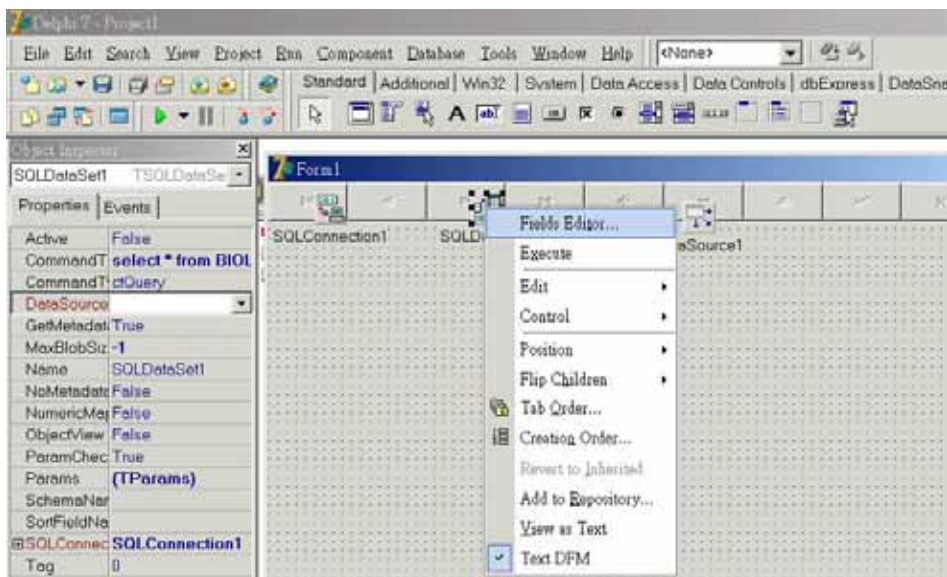


图1-8 激活TSQLDataSet的字段编辑器

请点击这个选项，Delphi便会显示一个空白窗口，请在这个空白窗口中再点击鼠标右键，Delphi便会显示一个快捷菜单，点击这个菜单中的 Add all fields选项，以便将代表BIOLIFE数据表中每一个字段的字段对象加入应用程序中，如图 1-9所示。接着Delphi便会把所有的字段对象加入刚才的空白窗口中，现在我们就可以点击这个窗口中的每一个字段对象，然后使用对象检视器来设置它们的特性值。

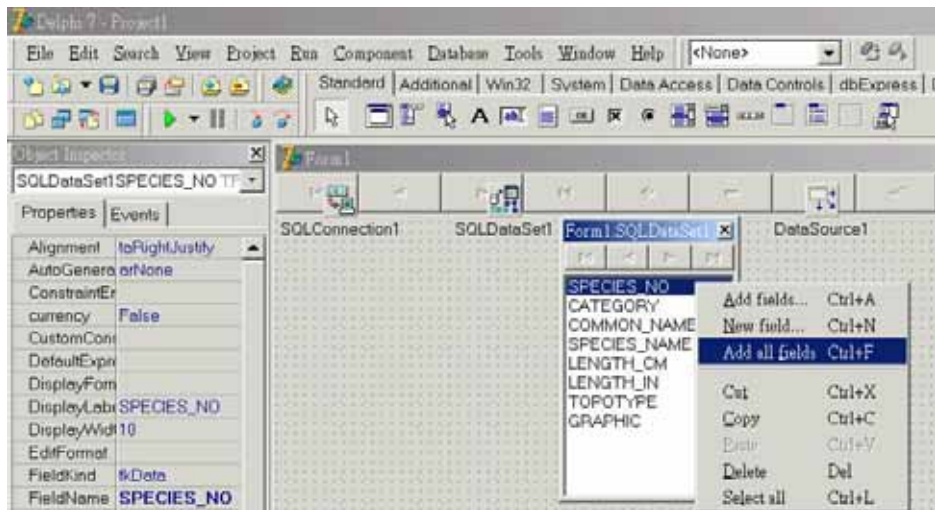


图1-9 加入所有的字段对象

现在我们希望应用程序在执行时显示中文的字段名称，而不是数据表中字段的英文名称，那么我们可以点击图 1-9中的每一个字段对象，然后在对象检视器中设置字段对象的 DisplayLabel 特性，接着输入这个字段的中文名称即可。例如图 1-10便是设置 SPECIES_NO 这个字段的中文名称的画面。

在我们一一设置完每一个字段对象的 DisplayLabel 特性值之后，便可以选择所有的字段对象，然后把它们拖曳到主窗体中，那么 Delphi 便会自动地在主窗体中产生能够适当显示每一个字段对象的数据感知组件，并且在这些数据感知组件中显示数据。例如图 1-11便是把图 1-10中的所有字段对象拖曳到主窗体中后的画面。

现在我们已经完成了第一个使用 dbExpress 组件实现的范例数据库应用程序，请执行它，此时我们便可以看到类似图 1-12的画面。dbExpress 组件果然可以顺利地 from InterBase 中访问数据，并且显示在数据感知组件中，就和 Delphi 原本的 BDE/IDAPI 组件一样方便。

但是请读者仔细观察图 1-12的画面，读者会发现图 1-12中的 TDBNavigator 中所有与修改数据有关的按钮都被暂停使用，例如代表修改数据的“▲”按钮和新增数据的“+”按钮。这意味我们无法使用这个范例应用程序来修改 BIOLIFE 数据表中的数据，也代表由 dbExpress 组件开发的数据库应用程序是无法修改的。

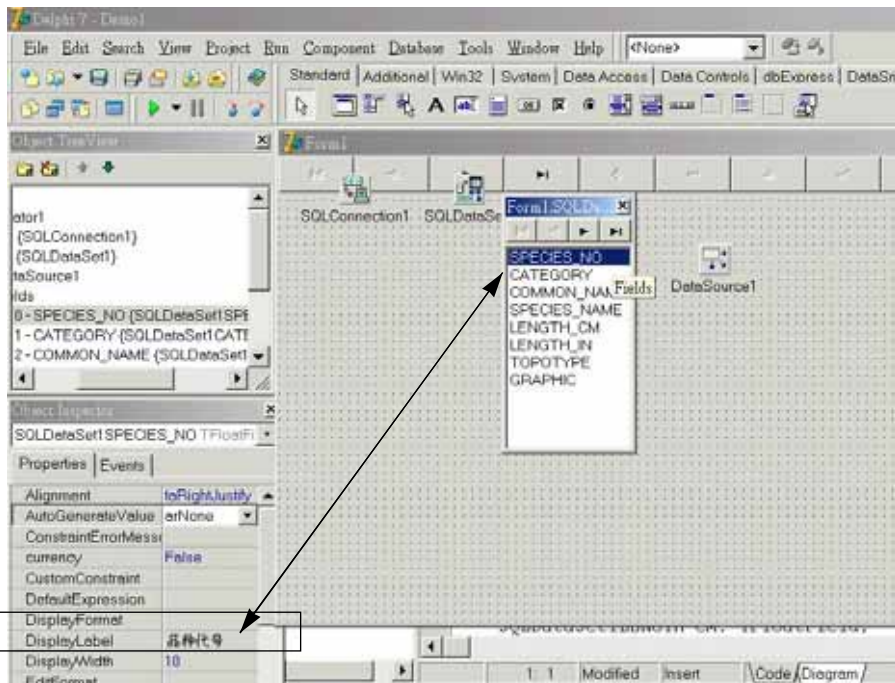


图1-10 使用对象检视器设置字段对象的特性值

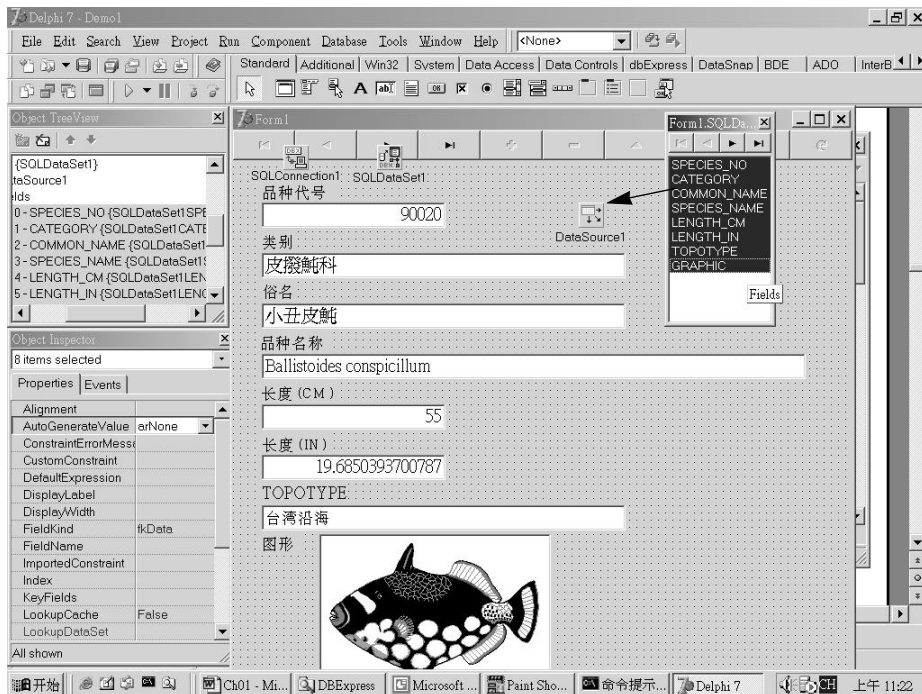


图1-11 将所有字段对象拖曳到主窗体中

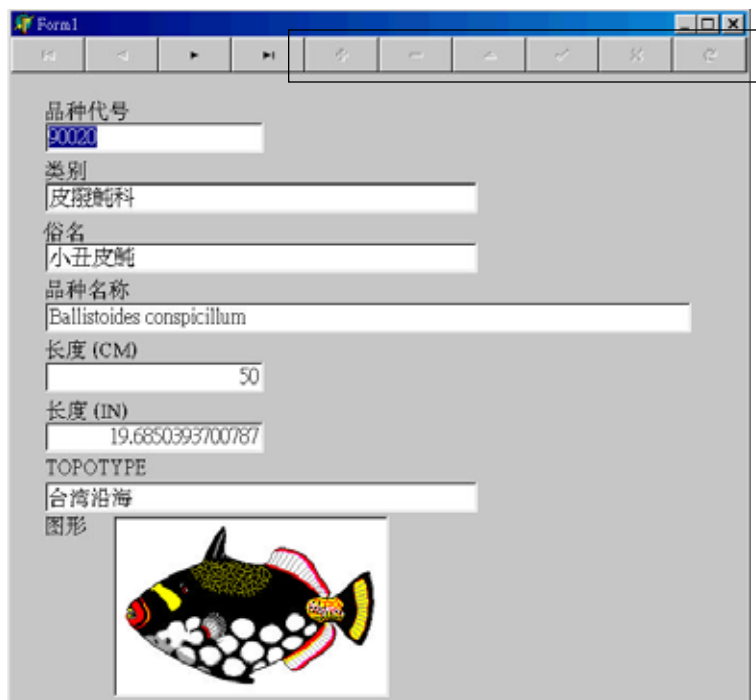


图1-12 范例应用程序执行的画面，请注意 TDBNavigator中所有与修改数据有关的按钮都被禁用，代表应用程序无法修改数据

现在如果读者点击向下一个记录的按钮移动到随后的记录，接着再点击往前的按钮欲把目前的记录移动到前一个记录时，此时范例应用程序便会显示下面的错误消息。

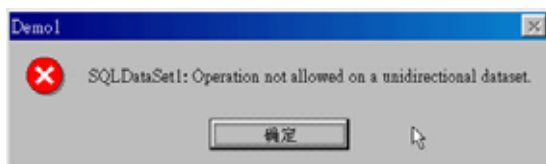


图1-13 点击TDBNavigator中的前一个记录的按钮时，范例应用程序会发生错误

这个错误消息的意思是由 dbExpress组件开发的应用程序使用只能向后走的单向游标，而无法向前走回到前面的记录。

也许读者会觉得很奇怪，既然 dbExpress无法修改数据，也无法任意移动目前记录的位置，那么 dbExpress不是一点用处都没有吗？如何能够用它来开发实际的数据库应用程序呢？

当然不，这是因为 dbExpress是使用与以往 BDE组件不一样的方式来设计的，因此如果读者按照使用 BDE的概念来使用 dbExpress便会觉得奇怪。事实上，dbExpress不但能够做到和 BDE组件一样的功能，甚至还有比 BDE更多的功能，在性能上也胜

过BDE。在本书进一步说明如何使用 dbExpress修改数据之前，先说明 dbExpress组件的一些重要的概念以及它的设计结构，如此一来读者将会更清楚地了解 dbExpress的设计原理，在稍后的章节中也将更能掌握 dbExpress。

1.3 使用dbExpress的概念

dbExpress组件组中与访问数据有关的组件，例如 TSQLDataSet、TSQLTable等都是从TDataSet类继承下来的，因此它们提供了所有和 TDataSet一样的功能，也可以和Delphi的数据感知组件使用在一起以开发数据库应用程序。图 1-14显示了这些 dbExpress组件的类结构图。

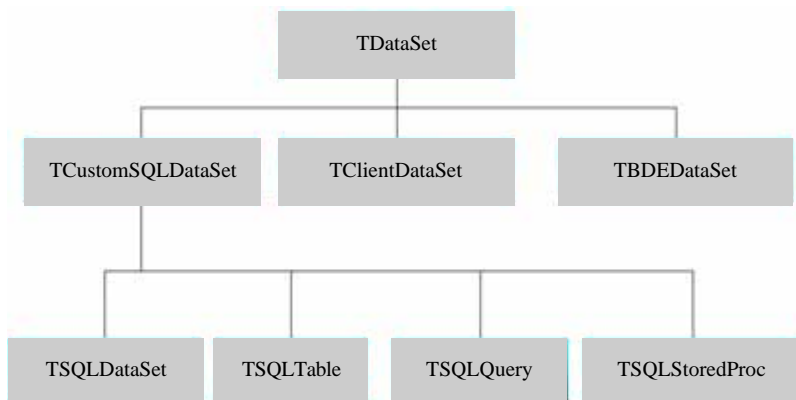


图1-14 dbExpress的类结构图

虽然 TSQLDataSet 等组件是从 TDataSet 继承下来的，但是它们与其他也从 TDataSet 继承下来的组件（例如 TBDEDataSet 等组件）不一样的地方是，通过这些 dbExpress 组件取得的结果数据集是只读的。这也就是说由这些组件取得的数据是不能修改的，而且由这些 dbExpress 组件取得的结果数据集的数据访问方式是只能由前往后访问，而无法由后往前访问。

如果程序员想能够修改由这些 dbExpress 取得的数据，或是想在结果数据集中以任意的方式移动目前记录的位置，那么程序员可以使用两种方法来达成：

- 1) 在应用程序中再搭配使用组件面板中 Data Access 选项卡中的 TDataSetProvider 和 TClientDataSet 组件。
- 2) 使用 dbExpress 组件面板中的 TSimpleDataSet 组件。

本书稍后将会同时说明如何使用这两种方式，现在先让我们说明如何使用同样位于组件面板 dbExpress 选项卡中的 TSimpleDataSet 组件。

TSimpleDataSet 组件是 Delphi 7 用来帮助程序员通过 dbExpress 组件修改数据的组件。基本上，使用 TSimpleDataSet 组件等同于使用 TSQLDataSet 组件加

TDataSetProvider组件，再加上 TClientDataSet组件。TSimpleDataSet不但能够访问数据，更能够让程序员修改数据和移动目前记录的位置。

基本上，TSimpleDataSet组件是使用 TSQLDataSet组件从后端数据源中取得数据，再通过内部的缓存（cache）机制管理数据，通过内部缓冲（buffering）机制允许程序员任意移动目前记录位置，并且在程序员需要将修改的数据写回数据源时，根据它内部维护的信息自动地帮程序员产生修改数据的 SQL语句，再通过 TSQLDataSet组件使用这些自动产生的 SQL语句把修改的数据更新回数据源。因此 TSimpleDataSet组件等于是帮程序员编写管理数据以及把修改的数据更新回数据源的程序代码，而无需程序员自己编写这些程序代码。

图1-15就是 TSimpleDataSet组件的类结构图，从图中可以看到 TSimpleDataSet是从包含缓存机制的 TCustomClientDataSet组件继承下来的。

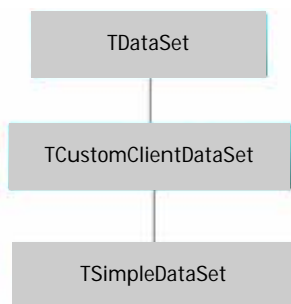


图1-15 TSimpleDataSet的类结构图

当程序员使用 TSimpleDataSet时就不需要再搭配使用 TSQLConnection和 TSQLDataSet，因为 TSimpleDataSet自己会在内部建立暂时的 TSQLConnection组件、TDataSetProvider组件以及一个 TInternalSQLDataSet组件。TSimpleDataSet会使用内部建立的 TSQLConnection连接至数据库，再使用内部建立的 TInternalSQLDataSet和 TDataSetProvider处理数据。由于 TSimpleDataSet内部使用了 TDataSetProvider组件，因此提供了修改数据的能力，比使用 TSQLDataSet只能提供数据查询的能力方便多了。

此外，虽然程序员在 Delphi 7 中是使用 dbExpress组件访问和处理数据，但是在 dbExpress组件之下，程序员仍然必须拥有每一个特定数据库的 dbExpress驱动程序才可以访问特定的数据库。当程序员像图1-5那样使用 TSQLConnection组件指定连接特定的数据库时，dbExpress便会加载与此数据库相关的 dbExpress驱动程序以访问这个数据库。

但是加载的 dbExpress驱动程序仍然需要调用特定数据库的客户端驱动程序，才能够真正访问特定数据库。例如，要使用 dbExpress访问 Oracle，那么除了需要 dbExpress中用于访问 Oracle的 DBEXPORA.DLL驱动程序之外，由于 DBEXPORA.DLL直接调用 Oracle的 Oracle Call Interface，因此还需要 Oracle客户端的 OCI.DLL，否则 dbExpress仍然无法访问 Oracle数据库。

例如，图 1-16 就是 dbExpress 在访问数据库时实际的结构。当 dbExpress 组件要访问 Oracle 时，就使用刚才描述的流程。例如 dbExpress 要访问 InterBase 时，就需要加载 dbExpress 中用于访问 InterBase 的 DBEXPINT.DLL 驱动程序以及 InterBase 客户端的引擎 GDS32.DLL。

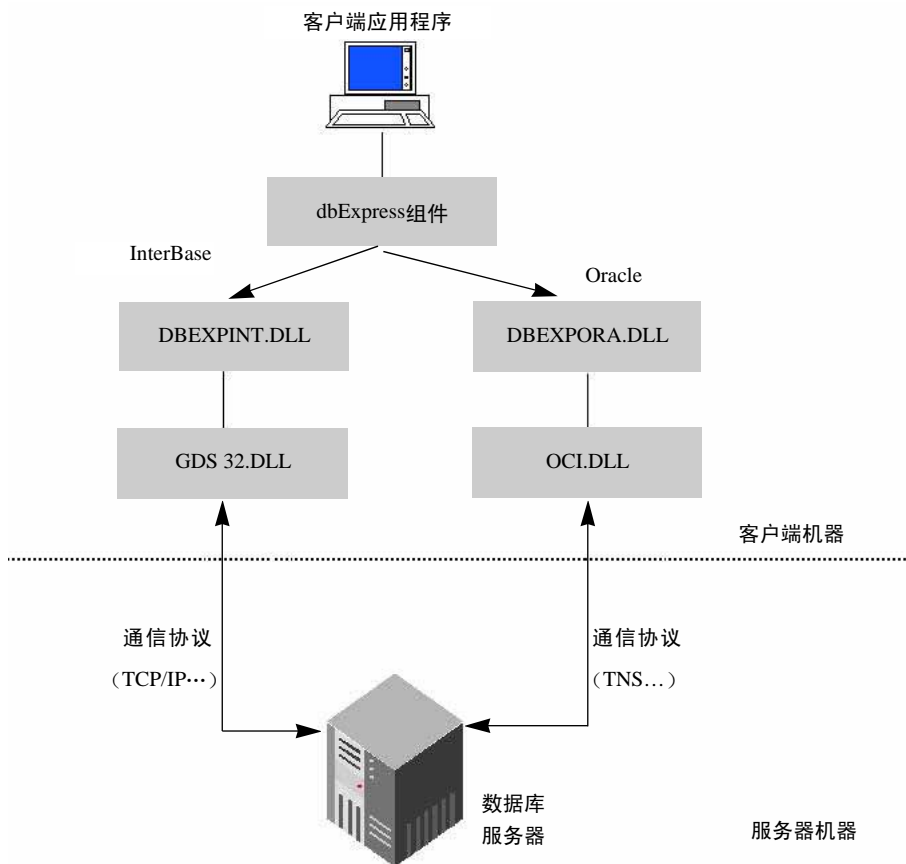


图 1-16 TSQLClientDataSet 的类结构图

在 dbExpress 组件通过驱动程序取得了结果数据集之后，程序员可以再使用 TSimpleDataSet 来修改结果数据集中的数据，因为 TSimpleDataSet 包含了内部缓存的机制，可以自动帮助程序员完成这些工作。如果我们更详细地说明什么是 TSimpleDataSet 内部的缓存机制，那么答案就更清楚了，这个缓存机制事实上就是以往 Delphi 中的 MIDAS 技术。Delphi 7 中的 MIDAS 已经是第 5 个版本了（Delphi 6 中的 MIDAS 是第 4 个版本），它在 Delphi 6 中被重新命名为 DataSnap。Delphi 7 中的 DataSnap 已经从以往的分布式应用系统结构技术转换为跨平台的标准数据访问技术。这个意思是说，MIDAS 5 不但可以用来开发分布式应用系统，在 Delphi 7 中也成为开发主从结构、数据库和单机应用程序的标准数据访问技术，并且能够同时在

Windows平台的Delphi 7中以及Linux平台的Kylix中使用。因此 DataSnap已经成为 Delphi的核心技术之一，所有的 Delphi程序员都必须了解并且纯熟地掌握 DataSnap技术，当然，本书也会充分地说明如何善用 DataSnap技术。

因此在 Delphi 7 中开发数据库应用程序的结构就如图 1-17所示的一样，应用程序通过dbExpress访问数据，再由DataSnap（MIDAS 5）来管理和修改数据。

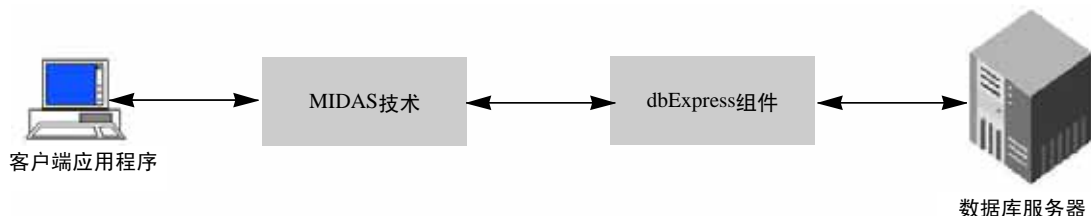


图1-17 TSQLClientDataSet的类结构图

图1-18是更详细的结构图， Delphi 7应用程序通过 dbExpress组件访问数据，而 dbExpress使用SQL语句从数据源中取得数据，当数据源根据 SQL语句产生结果数据集并且返回给dbExpress组件之后，dbExpress组件会再把结果数据集交由 DataSnap技术管理，以便程序员能够对结果数据集中的数据进行处理和修改。

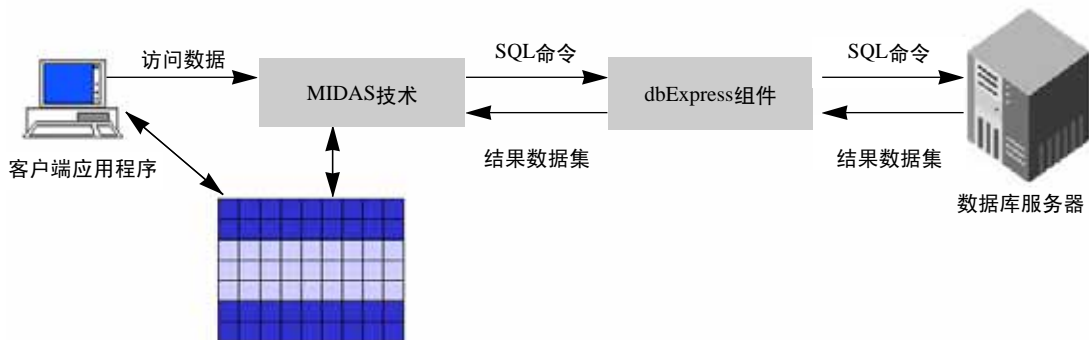


图1-18 TSQLClientDataSet的类结构图

从图1-18中我们也可以了解到，通过 TSimpleDataSet取得的数据事实上就是由 DataSnap在缓存内存中维护的数据。几乎所有通过 TSimpleDataSet的方法或特性值处理的数据都是存储在这些缓存内存中的数据，只有当程序员真的需要把缓存内存中的数据更新回数据源时，DataSnap才会通过SQL语句把经过修改的数据更新回数据库，在稍后的小节中会说明如何修改数据并且把数据真正更新回数据源中。

1.4 使用dbExpress修改数据

现在本书已经说明了 dbExpress的结构和原理，在本书随后的章节中会逐步详细说明dbExpress技术的精髓，让程序员能够精确地掌握 dbExpress技术。不过在学习

dbExpress的其他功能之前,先让我们以实际的范例来说明如何使用 dbExpress任意地移动记录位置和修改数据。

在本小节中也将同时说明如何使用 TSimpleDataSet以及使用 TSQLDataSet搭配 TDataSetProvider和TClientDataSet组件,以便证明在前面小节中提到的修改数据的方式。首先让我们延续前面 1.2小节的范例来说明如何完整地处理 CHINESEDEMO的 BIOLIFE数据表中的数据。第一步是让 1.2小节中使用的范例程序搭配 TDataSetProvider和TClientDataSet组件。

1.4.1 使用TSQLDataSet搭配TDataSetProvider和TClientDataSet组件

请回到Delphi 7集成开发环境,开启范例应用程序的主窗体,然后在主窗体中加入组件面板 Data Access选项卡中的 TDataSetProvider和TClientDataSet组件,最后再加入dbExpress选项卡中的 TSQLDataSet组件,设置 TSQLDataSet组件的 SQLConnection特性值为窗体中的 SQLConnection1,再设置它的 SQL特性值为 Select * from BIOLIFE,如图 1-19所示。

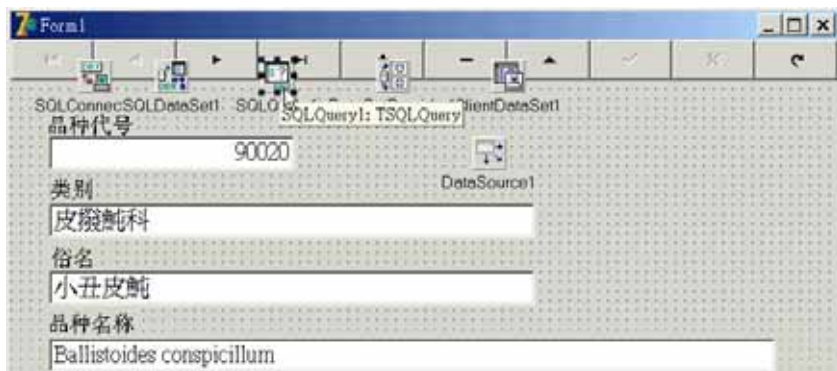


图1-19 在范例主窗体中加入 TDataSetProvider以及TClientDataSet组件

在前一小节中,本书说明了由 dbExpress取得的结果数据集虽然无法修改,但是只要搭配 Delphi的 DataSnap技术就能够允许应用程序修改其中的数据。因此现在我们要做的就是范例应用程序中加入 DataSnap的功能。

在加入了 TDataSetProvider组件之后,使用对象检视器设置它的 DataSet特性值为主窗体中原先的 TSQLDataSet组件,如图 1-20所示。

TDataSetProvider组件能够把属于 TDataSet类的组件(例如 TSQLDataSet,请参考图 1-14的类结构图)输出给 TClientDataSet组件,以便让 TClientDataSet组件能够管理输出的 TDataSet组件中的结果数据集数据。

接着使用对象检视器设置刚才加入的 TClientDataSet组件的 ProviderName特性值为主窗体中的 TDataSetProvider组件,如图 1-21所示。

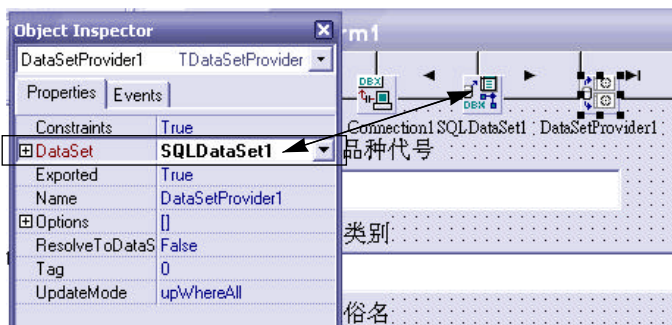


图1-20 设置TDataSetProvider组件的DataSet特性值

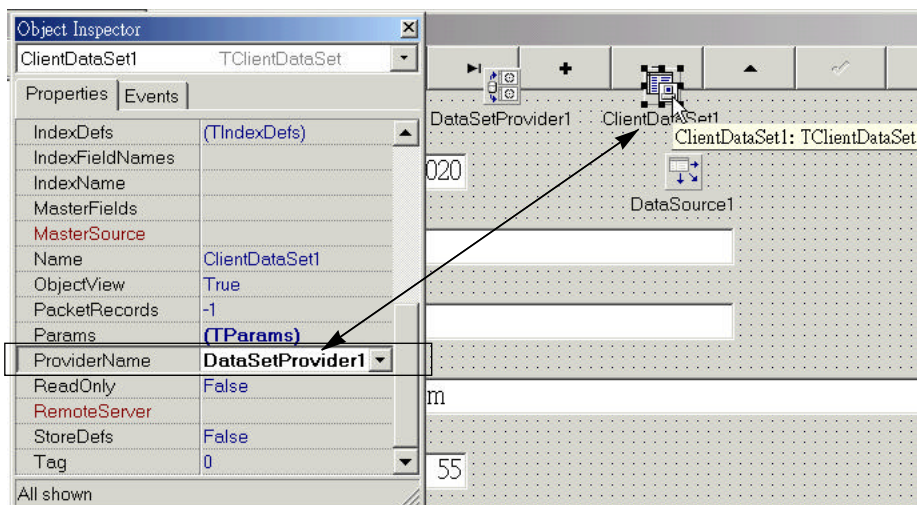


图1-21 设置TClientDataSet组件的ProviderName特性值

现在 TClientDataSet 组件便可以管理由 TDataSetProvider 输出的数据，而 TDataSetProvider 输出的数据就是它连接的 TSQLDataSet 从后端数据源中取得的结果数据集。

最后把主窗体中的 TDataSource 组件的 DataSet 特性值从原先的 TSQLDataSet 改为刚才加入的 TClientDataSet 组件，再把 TClientDataSet 组件的 Active 特性值设置为 True，如此一来 TClientDataSet 便从 TDataSetProvider 取得数据，并且显示在主窗体中的数据感知组件中。

现在请执行这个范例应用程序，读者会看到类似图 1-22 的画面，从图 1-22 中可以看到在使用了 TDataSetProvider 和 TClientDataSet 之后，范例应用程序便可以使用 TDBNavigator 任意地移动目前的记录，再也不会出现错误了。因此使用 DataSnap 技术之后，dbExpress 单向游标的限制便已经被克服了。

此外 TDBNavigator 中的新增、修改和删除按钮似乎也可以工作了，不像图 1-12 中

一样被暂停使用。读者可以在范例应用程序中试着修改或是删除数据，然后点击“~”按钮把数据更新回去。这样做似乎是正确的，但是如果读者结束范例应用程序，然后再次执行范例应用程序，会发现原先修改或是删除的数据又会出现范例应用程序中，这代表刚才修改和删除的动作并没有真正对数据发生作用，这是为什么呢？

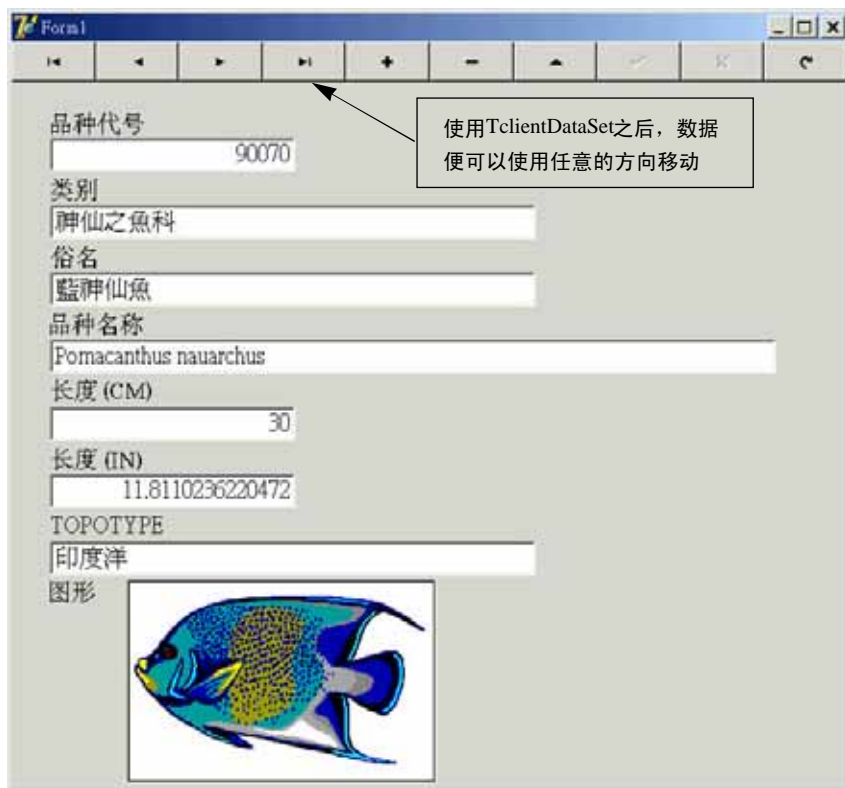


图1-22 执行修改过的范例程序

这是因为在使用 DataSnap 的应用程序中，当应用程序修改数据并且调用 TClientDataSet 的 Post 方法或是点击 TDBNavigator “~” 按钮更新数据时，事实上只是把数据更新回图 1-18 中由 DataSnap 管理的缓存内存中，数据并没有真正更新回后端数据源中。要真正地把修改的数据更新回数据源中，程序员必须再调用 TClientDataSet 的 ApplyUpdates 方法。

TClientDataSet 的 ApplyUpdates 方法会把图 1-18 中由 DataSnap 管理的缓存内存中所有已经被修改的数据（包括新增、修改和删除的数据）一起更新回后端数据源中。TClientDataSet 事实上是通过它连接的 TDataSetProvider 组件自动产生 SQL 语句帮助程序员更新数据的，在稍后的章节中会详细地说明这个流程。下面是 ApplyUpdates 方法的声明原型：

```
function ApplyUpdates (MaxErrors: Integer; Integer; virtual;
```


ApplyUpdates方法接受一个整数类型的参数，MaxErrors。MaxErrors代表当TDataSetProvider自动更新数据时，程序员所允许发生的错误次数。刚才本书已经说明了，当调用ApplyUpdates方法时，DataSnap会一起更新在缓存内存中应用程序从上次调用ApplyUpdates方法之后到这次调用ApplyUpdates方法之间被修改的所有数据，因此这可能会有许多修改的数据，而MaxErrors就代表在更新这些数据时程序员允许发生错误的次数。如果ApplyUpdates在更新数据时发生了超过MaxErrors指定的数量的错误，那么这整个更新动作便会被回滚。相反的，如果发生的次数小于或是等于MaxErrors，那么成功更新的数据仍然会被更新到数据源中，至于没有成功更新的数据则可以让程序员通过错误事件处理函数来决定如何处理这些失败的数据。本书在稍后讨论异常处理的章节中会详细说明这个问题。通常传递给ApplyUpdates方法的MaxErrors参数是0，代表不允许发生任何更新错误。或是传递-1，代表不管发生多少错误都没有关系，先把能够成功更新的数据更新回数据源中。

因此，现在要真正把范例应用程序修改的数据更新回数据源中便非常简单了，只需要在范例应用程序中调用TClientDataSet的ApplyUpdates方法即可。现在请在主窗体中加入一个TButton，设置它的Caption特性值为ApplyUpdate，如图1-23所示。

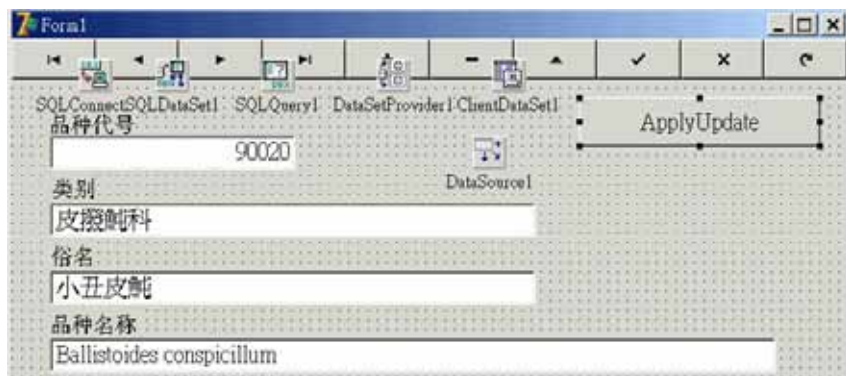


图1-23 在范例主窗体中加入TButton组件

接着在TButton的OnClick事件处理函数中编写如下的程序代码：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ClientDataSet1.ApplyUpdates(0);
end;
```

上面的程序代码调用了TClientDataSet的ApplyUpdates方法，并且传入0代表不允许发生任何错误，如果在修改数据时发生了任何问题，那么DataSnap便会产生异常错误让程序员得以处理错误状况，在稍后的章节中会说明如何处理DataSnap的异常状况。

请再次执行范例应用程序，试着修改数据，然后点击主窗体中的ApplyUpdate按

钮，读者便会发现现在数据真的被更新回后端数据源中了。

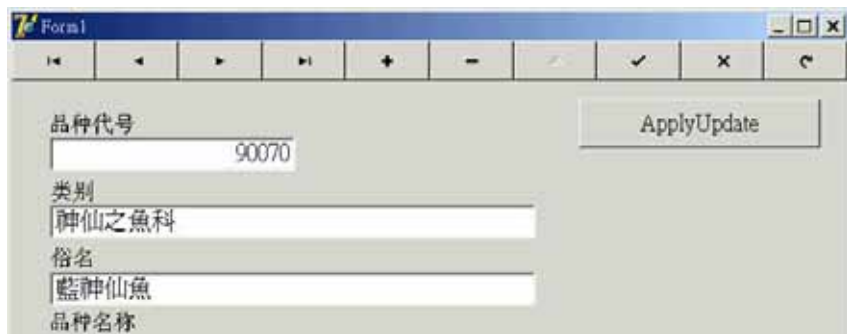


图1-24 范例程序现在可修改数据了

当然，如果读者希望用户无需额外点击按钮便能够把数据更新回数据源中，那么读者可以在 TClientDataSet 的 AfterPost 事件处理函数中直接调用 TClientDataSet 的 ApplyUpdates 方法，如下所示：

```
procedure TForm1.ClientDataSet1AfterPost (DataSet: TDataSet);  
begin  
    ClientDataSet1.ApplyUpdates (0);  
end;
```

不过这样写应用程序会降低性能，读者可以让 TClientDataSet 的修改数据达到一定的数量之后再调用 ApplyUpdates 方法，一次更新所有的数据。如此做比较有效率，在稍后的章节中会说明如何实现这种功能。

通过这个范例，读者可能认为使用 dbExpress 和 DataSnap 的应用程序似乎比较麻烦，然而事实上并非如此，因为在下一小节讨论 TSimpleDataSet 时读者便会知道我们可以通过 TSimpleDataSet 简化这些工作。

使用 dbExpress 和 DataSnap 比传统的开发方式有许多的优点，例如这样做比传统使用 BDE 的应用程序有更好的性能，也可以轻易地把应用程序改为分布式的 N 层应用系统，也可以把应用程序移植到 Linux 上。这些好处都是传统的开发方式无法轻易达成的，在读者熟悉了 dbExpress 和 DataSnap 之后，可能就不再想使用传统的数据库开发方式了。

1.4.2 使用 TSimpleDataSet 组件

在前一小节中讨论了使用 TSQLQuery 加上 TDataSetProvider 和 TClientDataSet 组件开发应用程序的方式。由于在使用 dbExpress 开发数据库应用程序时一定需要使用这些组件的组合，因此 Delphi 7 的 DataSnap 组件面板便提供了一个新的组件 TSimpleDataSet 来帮助程序员简化使用 dbExpress 开发数据库应用程序的步骤。

简单的说， TSimpleDataSet 组件就等于 TSQLQuery 加 TDataSetProvider 和

TClientDataSet组件，因此程序员只需要使用 TSQLClientDataSet一个组件便可以开发能够修改数据的应用程序。现在来说明如何使用 TSimpleDataSet组件。

首先点击Delphi的File|New|Application菜单建立一个新的Delphi应用程序，然后按照1.2小节说明的方式连接InterBase数据库。在主窗体中放入位于dbExpress选项卡的TSimpleDataSet组件，请读者注意图1-25显示了TSimpleDataSet已经在内部自动建立了TSQLDBConnection和TDataSet组件，并且将它们命名为 InternalConnection和 InternalDataSet，这样程序员就可以直接使用 TSimpleDataSet的InternalConnection和 InternalDataSet组件连接和处理数据，而不需要再额外使用 TSQLDBConnection和 TSQLQuery了。

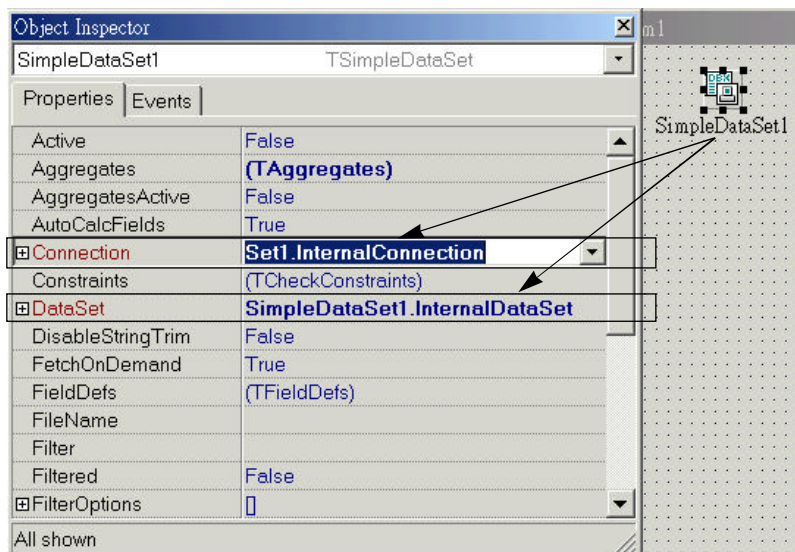


图1-25 TSimpleDataSet组件会自动在内部建立 TSQLConnection和DataSet组件

现在我们可以直接在对象检视器中展开 TSimpleDataSet的InternalConnection和 InternalDataSet的特性值并且设置到 CHINESEDEMO 数据库的连接，再将 TSimpleDataSet的DataSet\CommandText特性值设置为从 BIOLIFE数据表中取得数据，如图1-26所示。最后再将 TSimpleDataSet的Active特性值设置为 True以便从数据源中取得数据。

接着在主窗体中放入 TDataSource组件，将它的 DataSet特性值设置为刚才加入的 TSimpleDataSet组件，放入 TDBNavigator组件，将它的 DataSource特性值设置为刚才加入的 TDataSource。最后双击 TSimpleDataSet组件激活它的字段编辑器，加入所有的字段对象，再拖曳这些字段对象到主窗体中以建立数据感知组件。

最后，在主窗体中同样加入一个 TButton，将它的 Caption特性值设置为 ApplyUpdate，并且在它的 OnClick事件处理函数中调用 TSimpleDataSet组件的

ApplyUpdates方法，把修改的数据更新回数据源中。

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    SimpleDataSet1.ApplyUpdates(0);  
end;
```

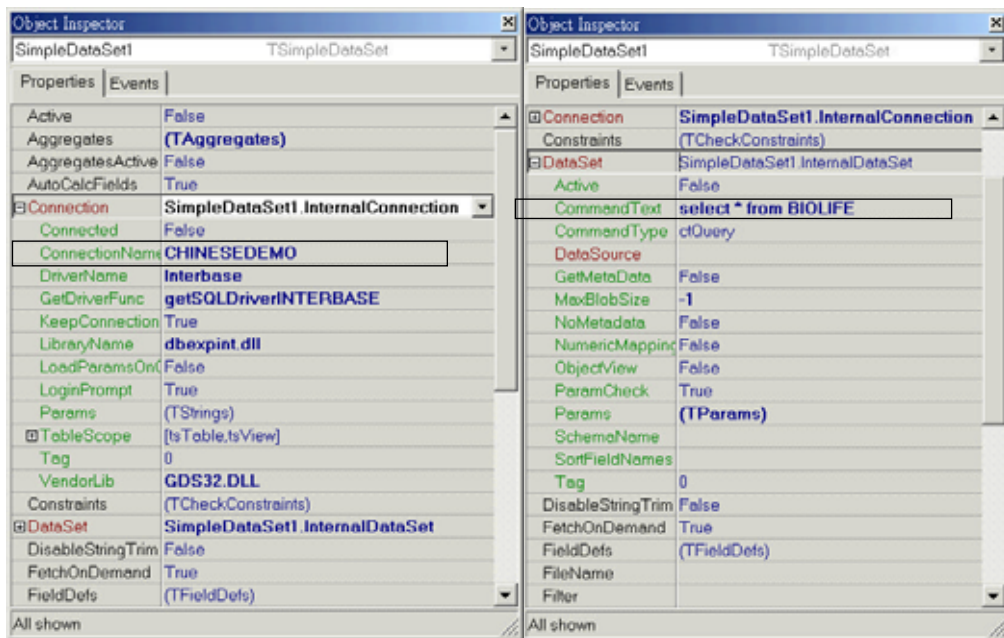


图1-26 使用对象检视器直接设置 TSimpleDataSet 的 InternalConnection
和 InternalDataSet 的特性值

此时范例应用程序应该看起来如图 1-27 所示。

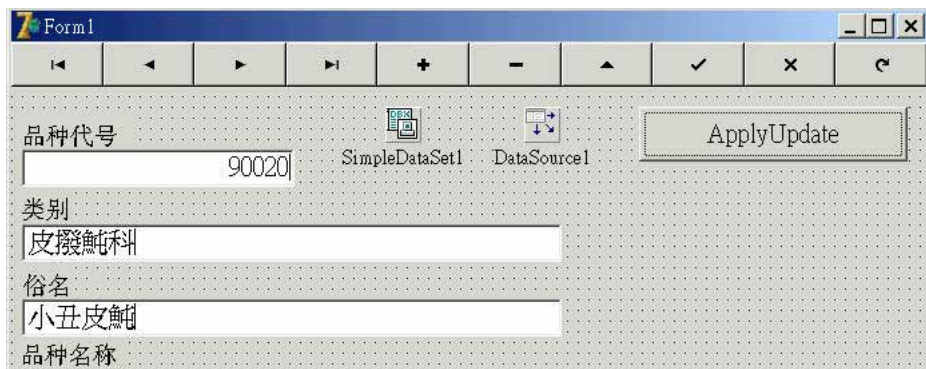


图1-27 执行使用 TSimpleDataSet 的范例程序

现在读者就可以执行这个使用 TSimpleDataSet 的范例应用程序了，图 1-28便是这

个新的范例应用程序执行的画面。这个新的范例应用程序与 1.3.1小节开发的范例应用程序在功能上是一模一样的，只是使用的组件不一样，使用 TSimpleDataSet比使用TSQLDataSet加TDataSetProvider和TClientDataSet组件方便得多，在开发一般应用程序的时候，直接使用 TSimpleDataSet是比较方便的。



图1-28 使用TSimpleDataSet范例应用程序执行的画面

也许读者会问，既然 Delphi 7提供的TSimpleDataSet组件可以更方便地开发数据库应用程序，那么还会需要使用 TSQLQuery加TDataSetProvider和TClientDataSet组件这种比较麻烦的方式吗？答案是肯定的，在后面讨论性能的章节中读者便会了解在高性能要求的应用中，TSQLDataSet加TDataSetProvider和TClientDataSet组件还是非常有用的。

TSimpleDataSet是Delphi 7才加入的新组件，在 Delphi 6和Kylix 2/3中的dbExpress则是使用 TSQLClientDataSet。Delphi 7使用 TSimpleDataSet代替 TSQLClientDataSet的原因除了TSimpleDataSet更容易使用之外，TSimpleDataSet的性能也比TSQLClientDataSet好了许多，比较适合用来开发简易或是2层的数据库应用系统，有关 TSQLClientDataSet效率的问题会在稍后的

章节中说明。但是如果读者已经在 Delphi 6中使用 TSQLClientDataSet组件或是需要开发跨平台的 dbExpress应用系统，那么仍然可以使用 TSQLClientDataSet。在 Delphi 7中，TSQLClientDataSet是放在 Delphi 7安装目录下的 \Demos\DB\SQLClientDataSet子目录中，读者可以自行将 TSQLClientDataSet安装到Delphi 7中再使用。

1.5 dbExpress驱动程序的设置

在前面 1.2小节中已经说明了 TSQLConnection组件使用的连接信息是来自 dbxconnections.ini这个文件的内容。当程序员使用 TSQLConnection建立一个定制的数据库连接时，例如前面建立的 CHINESEDEMO连接，TSQLConnection组件便在 dbxconnections.ini文件写入此定制连接的信息。事实上 dbExpress的连接信息是由两个配置文件（configuration file）定义的，它们分别是：

配置文件名称	功能说明
dbxconnections.ini	存储Delphi内建的数据库连接信息以及程序员定义的定制连接信息
dbxdrivers.ini	存储dbExpress支持的数据库连接信息。例如数据库 dbExpress驱动程序的进入点函数名称、用户名、密码、使用的 Transaction级别、数据库服务器名称以及数据库名称等，属于每一个数据库特定的信息

dbxdrivers.ini文件中定义的基本上是每一个特定数据库使用的连接信息。而当程序员在 TSQLConnection中建立新的数据库连接信息时，TSQLConnection会到 dbxdrivers.ini文件中找到此特定数据库的连接信息作为定制连接信息的模板（template），再由程序员修改其中特定的数值，例如用户登录名和密码等。因此我们可以说dbxdrivers.ini提供了每一个dbExpress支持的数据库的模板信息。

当程序员建立了定制的数据库连接之后，TSQLConnection便会在 dbxconnections.ini文件中写入此连接信息以便让 dbExpress驱动程序在实际连接时作为连接信息之用。例如，在前面我们建立的 CHINESEDEMO在dbxconnections.ini中存储了如下的信息：

```
[CHINESEDEMO]
DriverName=Interbase
BlobSize=-1
CommitRetain=True
Database=e:\LeeWei\Books\Delphi7\Datas\CHINESEDEMO.GDB
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
SQLDialect=1
Interbase TransIsolation=ReadCommitted
```

```
User_Name=sysdba  
WaitOnLocks=True
```

上面的连接信息中清楚地标明了 CHINESEDEMO是连接到InterBase数据库的定制连接，因此当dbExpress应用程序激活时看到此参数选项便会到 dbxdrivers.ini中搜寻InterBase的dbExpress驱动程序细节。而在dbxdrivers.ini中则有如下的信息：

```
[Interbase]  
GetDriverFunc=getSQLDriverINTERBASE  
LibraryName=dbexpint.dll  
VendorLib=GDS32.DLL  
BlobSize=-1  
CommitRetain=True  
Database=database.gdb  
Password=masterkey  
RoleName=RoleName  
ServerCharSet=ASCII  
SQLDialect=1  
Interbase TransIsolation=ReadCommitted  
User_Name=sysdba  
WaitOnLocks=True
```

从上面的信息中可以看到对于 InterBase的连接，dbExpress需要加载dbexpint.dll这个DLL文件，而且dbexpint.dll函数库的进入点是 getSQLDriverINTERBASE。dbExpress在使用LoadLibrary加载dbexpint.dll之后，就可以使用GetProcAddress取得getSQLDriverINTERBASE函数的地址，调用getSQLDriverINTERBASE就可以取得dbExpress的ISQLConnection接口，接着就可以连接到InterBase数据库了。在后面的第13章中会说明dbExpress的实现原理，读者可以参考其中说明的内容。而dbexpint.dll则调用了InterBase的客户端API GDS32.DLL函数库来真正与InterBase连接交互。

Delphi 7已经正式支持MS SQL Server 2000以及Informix，但是在前面图1-4中我们却只看到了DB2、InterBase、MYSQL以及Oracle数据库。那么我们如何建立连接到MS SQL Server 2000或是Informix的定制数据库连接呢？如果读者了解了刚才说明的dbxconnections.ini和dbxdrivers.ini这两个配置文件的功能，就可以知道这是非常容易的。

首先，读者可以开启dbxdrivers.ini这个dbExpress驱动程序模板文件，便会在文件一开始的地方看到如下的信息：

```
[Installed Drivers]  
DB2=1  
Interbase=1  
MYSQL=1  
Oracle=1
```

这四个数据库的标志都设置为 1，因此这四个数据库连接信息就是我们在图 1-29 中看到的dbExpress支持的四个数据库。因此要让 TSQLConnection也在Delphi的集成开发环境中允许程序员使用 MS SQL Server和Informix，我们只需要在 Oracle=1 下面加入如下二行程序代码即可：

```
Informix=1
```

```
MSSQL=1
```

接着存储 dbxdrivers.ini，再重新使用 TSQLConnection建立新的数据库连接，那么读者就可以看到如图 1-30所示的画面，现在我们也可以建立 MS SQL Server和 Informix的定制连接了。

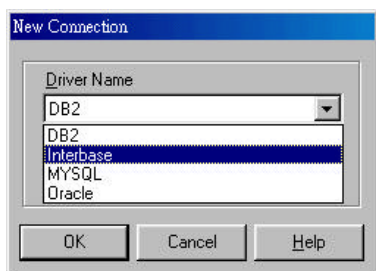


图1-29 在默认情况下TSQLConnection
只提供了四个数据库的连接信息

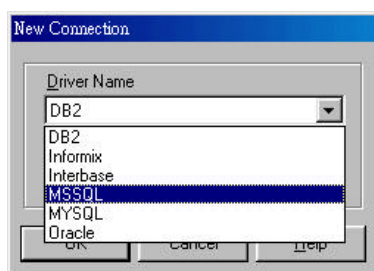


图1-30 在dbxconnections.ini中加入了MS SQL Server
和Informix的驱动程序标志之后便可在 TSQL-
Connection中看到新的内置数据库连接

如果点击图 1-30中的MSSQL，那么就可以在 dbExpress Connections对话框中看到 TSQLConnection从dbxdrivers.ini配置文件搜寻用于 MS SQL Server的连接模板信息（见图 1-31）。之后我们就可以修改其中的特性值以连接到正确的 MS SQL Server了。

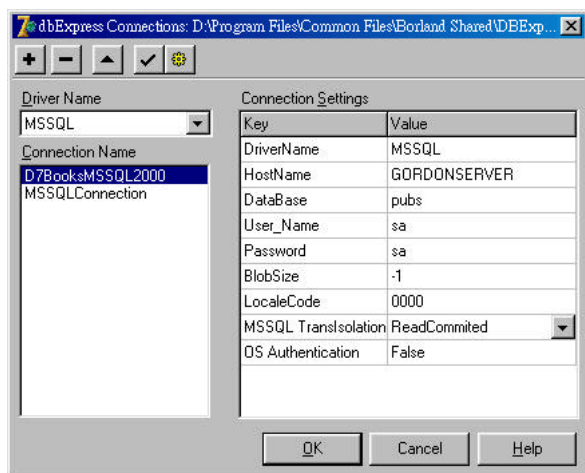


图1-31 设置使用MS SQL Server的定制数据库连接之后，TSQLConnection
直接从dbxdrivers.ini中取得MS SQL Server特定的连接信息

最后，我们开启 TSQLConnection 的 View Driver Settings 对话框，就可以清楚地看到每一个 dbExpress 驱动程序使用的特定数据库厂商的函数库。例如，从图 1-32 中我们可以确定 Delphi 7 的 MS SQL Server 驱动程序是直接使用 OLEDB 来连接，而不是使用 ADO。

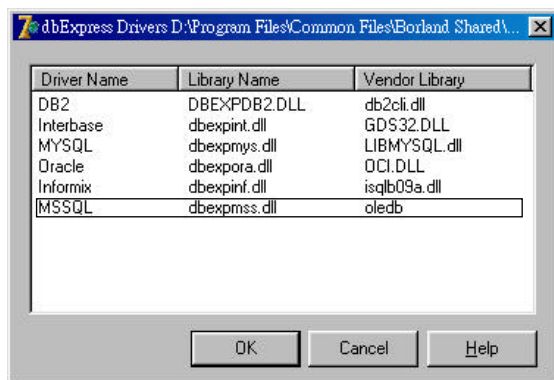


图1-32 在TSQLConnection的View Driver Settings对话框中现在可以看到各个dbExpress驱动程序使用的底层厂商驱动程序，例如 MS SQL Server是使用OLEDB连接MS SQL Server数据库的

其实Delphi也支持连接PostGreSQL的驱动程序，读者可以使用本小节讨论的过程为Delphi 7加入连接PostGreSQL的dbExpress信息。

1.6 结论

本章以范例的形式说明了如何使用 dbExpress 快速开发数据库应用程序，让读者能够马上使用 dbExpress 组件来编写应用程序。对于熟悉原先 BDE 组件的程序员来说，新的 dbExpress 技术似乎比较麻烦，需要使用数个组件才能开发出能够修改数据的数据库应用程序。但是通过使用新的 TSimpleDataSet 组件，dbExpress 开发模式就和以往的 BDE 一样方便。对于新接触 Delphi 的程序员来说，dbExpress 组件也是非常方便和直观的。

dbExpress 技术包含 dbExpress 组件、dbExpress 驱动程序以及 DataSnap 技术，本章讨论的内容只是 dbExpress 的冰山之一角，读者除了使用 dbExpress 组件开发应用程序之外，也需要熟悉许多技术以及如何巧妙地使用 TSimpleDataSet 的技术来处理数据。这些技术都会后面的章节中一一讨论。

本章除了说明如何使用 dbExpress 开发应用程序之外，也说明了 dbExpress 的原理和结构，让读者能够掌握 dbExpress 的核心概念，这有助于了解随后章节的内容。当然，在稍后的章节中会继续说明更多重要的 dbExpress 概念和技术。

第2章 使用dbExpress组件

在上一章中讨论了 dbExpress 的概念以及如何使用 dbExpress 快速建立应用程序。从本章开始将逐渐让读者了解更多有关使用 dbExpress 的细节以及如何更精确地使用 dbExpress 中的组件来开发应用程序。

由于完整的 dbExpress 技术需要集成 Delphi 的 MIDAS 技术，也就是从 Delphi 6/7 开始称为 DataSnap 的技术。因此在本章以及随后的章节中也将会一并讨论许多 DataSnap 技术。DataSnap 不但可以搭配 dbExpress 开发单机以及主从结构的应用系统，也可以结合 MTS/COM+ 等组件模型开发分布式多层应用系统，因此 DataSnap 在 Delphi 以及 Linux 上的 Kylix 中是非常重要的技术，读者一定要切实掌握这项技术。

现在就让我们从 dbExpress 中最重要的数据处理组件 TSimpleDataSet 开始学习 dbExpress 和 DataSnap 技术。

2.1 使用 TSimpleDataSet 组件

在第1章中，本书已经说明了在使用 Delphi 7 的 dbExpress 开发数据库应用程序时最基本的方式是使用 TSQLConnection 和 TSimpleDataSet 组件，如此一来才能够处理和修改数据，因此读者必须了解如何使用 TSimpleDataSet 组件才能正确地使用 dbExpress 技术。

TSimpleDataSet 组件有许多重要的特性和事件让程序员能够精确地控制处理数据的行为。TSimpleDataSet 的许多特性值都会影响它如何访问和处理数据，有一些特性值也和性能有重要的关系。因此这些特性是读者一定要了解和能够正确使用的，下面的小节便讨论了这些重要的特性以及相关的事件。

1. DataSet\CommandType

当程序员使用 TSimpleDataSet 组件时，必须要决定如何使用它来访问数据。TSimpleDataSet 组件使程序员可以执行各种 SQL 语句来访问和处理数据，也使程序员可以使用它来执行后端数据源中的存储过程 (Stored Procedure)，或是直接使用它来取得数据源中某一个数据表中的数据，因此 TSimpleDataSet 组件的功能可以说是非常丰富的。

这些决定如何使用 TSimpleDataSet 的特性便是 DataSet\CommandType，根据程序员为 DataSet\CommandType 设置的特性值可以决定 TSimpleDataSet 的执行行为。下面的表格列出了 DataSet\CommandType 可以设置的特性值以及这些特性值的意义：

CommandType特性	意 义
ctQuery	执行SQL语句
ctStoredProc	执行后端数据源中的存储过程
ctTable	访问指定的数据表中的所有数据

TSimpleDataSet的DataSet\CommandType的默认值是ctQuery，主要是执行SQL语句，如果程序员需要使用TSimpleDataSet执行存储过程，那么必须设置DataSet\CommandType为ctStoredProc。

2. Active特性

Active特性值可以让TSimpleDataSet组件执行程序员在DataSet\CommandText特性值中指定的SQL语句（当DataSet\CommandType是ctQuery时），或是取得DataSet\CommandText特性值指定的数据表中的所有数据（当DataSet\CommandType是ctTable时）。不过设置Active特性值只能让TSimpleDataSet执行会返回结果数据集的SQL语句，对于不返回结果数据集的SQL语句，程序员必须调用TSimpleDataSet的Execute方法。下面的表格总结了TSimpleDataSet组件设置Active特性值以及调用Execute方法的意义：

执行SQL语句	意 义
Active特性值	执行会返回结果数据集的SQL语句，例如Select...的SQL语句
Execute	执行不返回结果数据集的SQL语句，例如Delete、Update、Insert或是DDL语句

设置Active特性值为True就等于调用TSimpleDataSet的Open方法，设置为False就等于调用TSimpleDataSet的Close方法。

3. PacketRecords特性

TSimpleDataSet组件的PacketRecords是一个非常重要的特性值，因为它控制了TSimpleDataSet组件如何访问数据。TSimpleDataSet组件的PacketRecords基本上可以设置成下列表格中的数值。不同的PacketRecords特性值代表了不同的数据访问行为。

PacketRecords特性	意 义
-1	一次从后端数据源中取得所有数据
0	取得描述数据源的元数据信息
正数	一次只取得指定数量的记录

PacketRecords的默认值是-1，代表当TSimpleDataSet组件开启后它会一次把后端数据源中的所有数据读到客户端。对于数据量少的数据表来说这种访问行为没有什么关系。但是请想想，对于数据量大的数据表，例如拥有数万个记录的数据表，这样做不但没有效率，而且会造成沉重的网络负荷。这是程序员应该尽量避免的。

因此对于拥有大量数据的数据源来说，程序员应该设置PacketRecords特性值为一个正数，例如20。这就代表当TSimpleDataSet开启时它只会访问20个记录，就暂

时停止。如果用户浏览或处理到了 20个之后的记录，那么 TSimpleDataSet便会自动从后端数据源中取得下 20个记录。这种访问行为非常适合一般的使用，因为用户一次也不可能浏览或处理太多的数据，因此让 TSimpleDataSet以分段的方式访问客户端需要的数据不但可以加快应用程序的反应时间，减轻客户端的负荷，也可以降低网络的数据传递量。不过程序员也不能将 PacketRecords特性值设置为太小的数值，因为这会造成 TSimpleDataSet频繁地从后端数据源访问数据，反而会降低性能，增加网络的往返次数。一般来说，将 PacketRecords设置为 100左右的数值是比较好的，当然读者在实际应用中需要根据同时使用客户端的人数自行调整 PacketRecords数值，在稍后讨论性能的章节中会再说明。

最后，如果程序员将 PacketRecords设置为 0，那么就代表要从后端数据源中取得描述结果数据集的元数据。一般来说，当 TSimpleDataSet/TClientDataSet第一次开启后端的结果数据集时，都会先从数据源中取得元数据，建立客户端结果数据集的结构以便存储从后端结果数据集取得的数据。

所谓元数据（ metadata ）是指描述数据的信息。例如描述数据表的元数据包括了数据表所有的字段名称、字段类型、字段长度等信息。通常客户端能够根据这些元数据正确地建立客户端数据集，以及自动产生正确的 SQL语句。元数据对于程序是否能够有效率和正确地执行有很大的影响。

TSimpleDataSet/TClientDataSet会不会在第 1次开启后端数据源时取得元数据是根据不同的dbExpress驱动程序决定的。不过以目前 dbExpress驱动程序优化做得越来越好的状况来看， dbExpress驱动程序都避免进行这个动作以增加性能，在稍后的章节中会详细的讨论。

由于 PacketRecords特性值是如此重要，因此程序员在使用组件时必须对于这个特性值进行适当的设置。

4. Data特性

当 TSimpleDataSet从后端数据源取得数据之后，这些数据便存储在 TSimpleDataSet组件的 Data特性值中，因此程序员可以通过访问 Data特性值来取得原始数据。当客户端应用程序修改数据时，存储在 Data中的数据会被改变，同时所有被客户端应用程序修改的数据也都会暂时存储在 Delta特性值中。

5. Delta特性

TSimpleDataSet的 Delta特性值存储的是客户端应用程序对于 Data特性值中数据的修改。在 TSimpleDataSet从后端数据源取得数据时， Delta特性值的内容最初是空白的。但是一旦应用程序修改了数据，那么修改的数据便会暂时存储在 Delta中。而当

应用程序调用了 TSimpleDataSet 的 ApplyUpdates 方法之后, Delta 特性值中的数据便会真正更新回后端数据库中。在 ApplyUpdates 方法成功执行完毕之后, Delta 中的数值便会被清除。程序员可以通过访问 Delta 中的数值来取得目前被修改的数据。

虽然在一般的应用中程序员也许并不需要直接使用 Data 和 Delta 特性值,但是在许多高级的应用中这两个特性值却可以发挥非常大的作用,特别是在处理复杂的数据运算时,在稍后的章节中会有范例介绍如何使用 Data 和 Delta 特性。

考虑到更大的控制能力以及稍后章节会讨论的性能因素,因此笔者建议各位读者尽量不要直接使用 TSimpleDataSet 组件,而使用 TSQLQuery、TClientDataSet 和 TDataSetProvider 组件。当然,如果只是一般的应用,那么使用 TSimpleDataSet 是非常便利的。因此本小节随后的范例都将同时使用 TSQLClientDataSet 组件和这三个组件来说明。

TSimpleDataSet 组件还有许多重要的特性和事件,在稍后的章节中将会一一介绍这些方法、特性和事件。现在就让我们开始通过范例来学习如何使用 TSimpleDataSet 组件。

2.1.1 使用动态 SQL 语句处理数据

在第1章中已经说明了如何使用 TSimpleDataSet 组件,我们通过将 SQL 语句设置到 TSimpleDataSet 的 DataSet\CommandText 特性值中便可以从后端数据源中取得数据。本小节将进一步说明如何使用动态 SQL 语句来访问和处理数据。

步骤1: 建立数据模块和 dbExpress 组件

首先在 Delphi 集成开发环境中点击 File|New|CLX Application, 再点击 File|New|Other... 菜单, 在 New Items 对话框中点击 CLX Data Module 图标, 建立一个 CLX 数据模块, 如图 2-1 所示。

在空白的 CLX 数据模块中加入一个 TSQLConnection 组件, 将它的 Name 特性值设置为 scnnDemo。双击 TSQLConnection 组件以激活它的组件编辑器, 并且连接到 InterBase 范例数据库 D7Books.GDB, 如图 2-2 所示 (D7Books 使用 Add Connection 建立的 InterBase 连接, 请读者自行建立, 并且设置 ServerCharSet 特性值为空白)。

接着在 CLX 数据模块中放入一个 TSimpleDataSet 组件, 设置它的 Name 特性值为 scdsBooks, 设置它的 DBConnection 特性值为 scnnDemo。虽然 TSimpleDataSet 会自动在内部建立 TSQLConnection, 但是我们仍然可以强迫 TSimpleDataSet 使用外部的 TSQLConnection。

再设置 scdsBooks 的 DataSet\CommandText 特性值为 select * from BOOKS, 设置 Active 特性值为 True, 从 BOOKS 数据表中取得所有的数据。最后放入另外一个

TSimpleDataSet组件, 设置它的Name特性值为scdsPublishers, 设置DBConnection特性值为scnnDemo, 再设置DataSet\CommandText特性值为select * from PUBLISHERS where VID = :VID。从这个范例可以看到使用外部 TSQLConnection的好处, 因为在默认情况下scdsBooks和scdsPublishers都会在内部自行建立 TSQLConnection, 但是既然scdsBooks和scdsPublishers都连接到相同的数据库, 那么我们就需要建立两个TSQLConnection浪费资源。

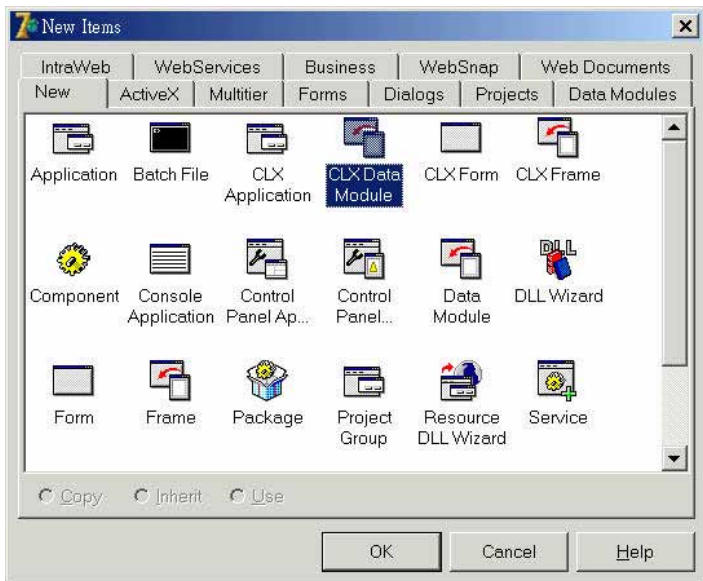


图2-1 建立CLX数据模块

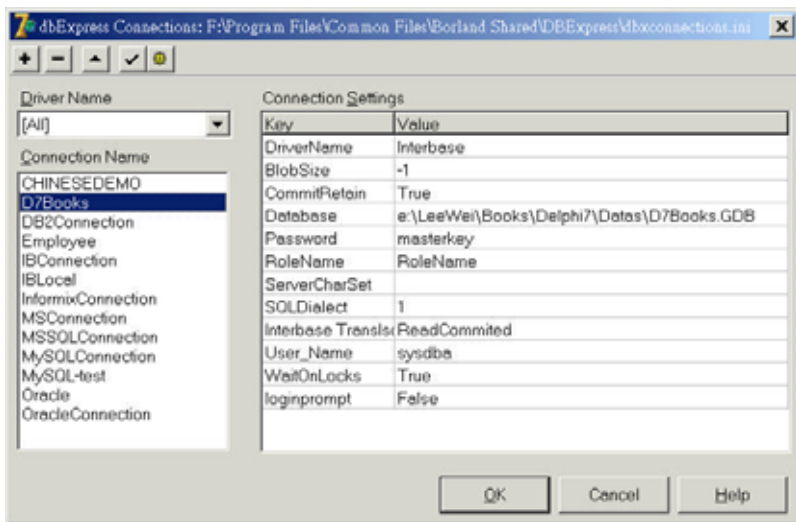


图2-2 激活TSQLConnection组件的组件编辑器

scdsPublishers在DataSet\CommandText特性值中使用的SQL语句称为动态SQL, 因为在这个SQL语句中定义了一个参数:VID。这个参数的数值会在应用程序执行时动态地传入, 再根据这个参数值从 PUBLISHERS数据表中取得VID字段拥有传入的参数值的数据。当然, 在这个SQL语句中参数名称VID可以命名为其他名称, 例如:ID1或是:PID。在SQL语句中动态参数是以:

:参数名称

形式定义的, 而且在SQL语句中可以使用多个动态参数。

dbExpress组件的TSQLDataSet、TSQLQuery、TSQLStoredProc和TSimpleDataSet都允许程序员使用有动态参数的SQL语句来访问数据。此时范例应用程序的数据模块看起来如图2-3所示。

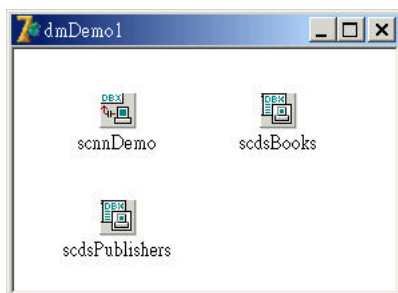


图2-3 范例应用程序的数据模块

由于scdsPublishers组件定义了动态参数, 因此它会自动分析SQL语句, 并且从其中取得所有动态参数, 然后存储在TSimpleDataSet组件的DataSet\Params特性值中。我们可以点击scdsPublishers组件, 然后在对象检视器中双击它的Params特性, 那么Delphi便会显示scdsPublishers定义的动态参数。例如, 图2-4便是双击scdsPublishers组件的Params特性之后, Delphi显示的参数对话框, 其中列出了VID这个动态参数。此时程序员可以点击这个动态参数, 然后在对象检视器中设置此动态参数的特性值。图2-4显示了VID是一个输入参数, 我们需要设置它的数据类型是字符串 (ftString)。

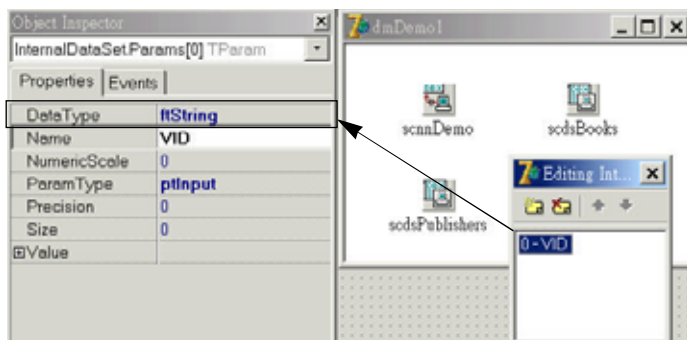


图2-4 scdsPublishers组件在Params特性值中定义的动态参数

步骤 2: 建立范例应用程序的主窗体

回到范例应用程序的主窗体, 在其中放入 TDataSource, 并且设置它的 DataSet 特性值为步骤 1 建立的 scdsBooks 组件 (注: 读者必须在主窗体中先 use 步骤 1 的数据模块, 即点击 File|Use Unit... 选项, 选择步骤 1 的 dmDemo1 数据模块), 再放入 TDBNavigator 和 TDBGrid 组件, 设置它们的 DataSource 特性值为刚才加入的 TDataSource。最后在主窗体下方放入一个 TPanel 组件, 此时范例主窗体看起来类似于图 2-5。



图2-5 范例应用程序的主窗体

现在我们希望当用户浏览一本书籍时, 能够同时显示出出版这本书的出版商信息, 因此我们需要显示 PUBLISHERS 数据表中的数据。请开启数据模块, 双击 scdsPublishers 组件以激活它的字段编辑器, 接着把 PUBLISHERS 数据表的所有字段对象拖曳到刚才加入的 TPanel 组件中。此时主窗体如图 2-6 所示。

接下来的工作就是实现此范例应用程序, 它需要的功能就是当用户移动书籍数据时, 把出版此书的出版商信息显示在下方的数据感知组件中。

如果读者建立 CLX 类型的应用程序, 那么可能无法以拖曳的方式建立图 2-6 中的数据感知组件, 读者需要自行加入 TDBEdit 组件来显示数据。

步骤 3: 实现范例应用程序

要实现浏览书籍时同时显示出出版商信息的功能, 那么我们首先需要根据目前正被浏览的书籍的出版商编号到 PUBLISHERS 数据表中搜寻。而在前面建立数据模块时

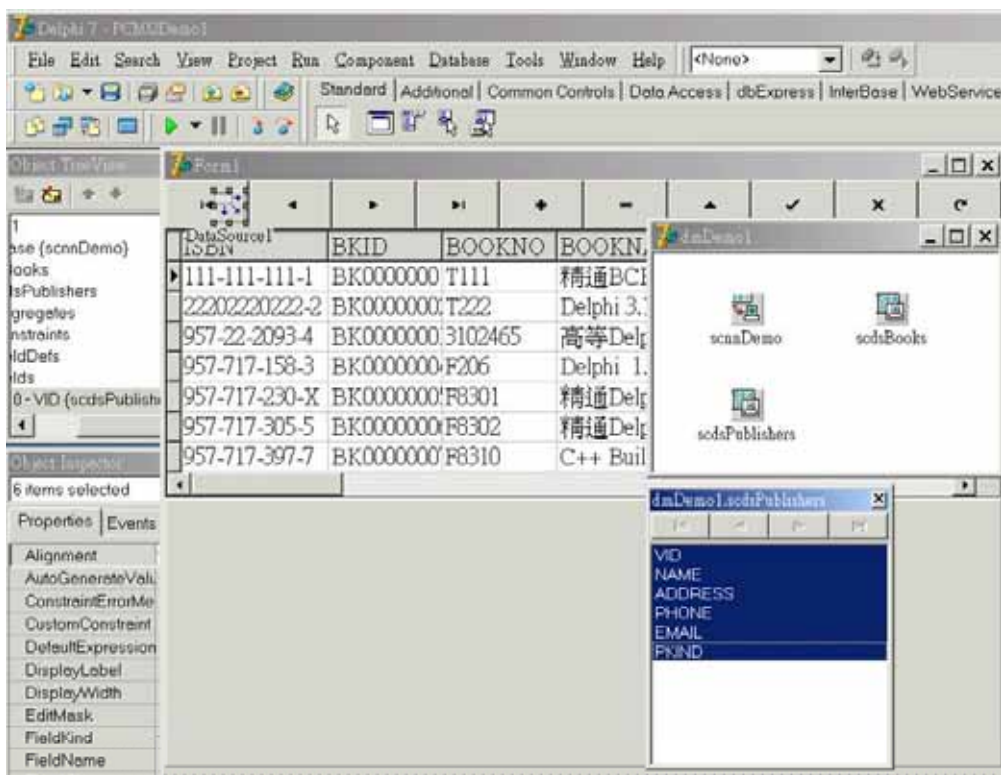


图2-6 在主窗体中加入显示scdsPublishers字段的数据感知组件

scdsPublishers组件的SQL语句正是根据出版商编号从 PUBLISHERS数据表中取得数据的, 因此我们只需要把目前正被浏览的书籍的出版商编号传递给 scdsPublishers组件即可。进行这个工作的好时机便是当用户将目前的记录位置移动到其他记录时, 因此我们可以在scdsBooks的AfterScroll事件处理函数中进行这个工作。

请在数据模块中点击 scdsBooks, 双击它的 AfterScroll事件, 并且编写如下的程序代码:

```
procedure TdmDynamicSQL.scdsBooksAfterScroll (DataSet: TDataSet;  
begin  
    try  
        Self.scdsPublishers.Active := False;  
        Self.scdsPublishers.DataSet.Params.ParamByName ('VID').Value :=  
            Self.scdsBooks.FieldByName ('VID').Value;  
        Self.scdsPublishers.Active := True;  
    except  
        on Exception do;  
    end;  
end;
```

上面的程序代码首先把目前书籍的出版商 ID传入到scdsPublishers定义的动态参数中。为此, 我们只需要访问 scdsPublishers的DataSet.Params特性, 再调用DataSet.Params返回的TParam对象的ParamByName方法即可。最后开启scdsPublishers即可取得正确的出版商数据。

步骤 4: 执行范例应用程序

现在就可以编译并且执行范例应用程序了, 图 2-7就是范例应用程序此时执行的画面。当用户使用窗体上方的 TDBNavigator移动数据时, 下方显示出版商的数据感知组件便会显示出正确的出版商数据。

现在我们已经使用 dbExpress组件和动态SQL语句的方式顺利地取得范例应用程序需要的数据了。接着让我们观察 dbExpress如何处理修改的数据。

步骤 5: 取得目前被修改的数据记录数信息

回到主窗体, 在窗体中加入一个 TBitBtn组件, 设置它的 Caption特性值为“ApplyUpdates”。再加入一个 TStatusBar组件, 双击它并且加入一个新的 TStatusPanel组件。接着我们希望主窗体也能够显示书籍的作者名称, 因此另外在主窗体中加入一个 TLabel组件, 设置它的Caption特性值为“作者”, 再加入一个 TDBEdit组件, 此时范例主窗体应该类似于图 2-8。

要在主窗体中显示书籍的作者, 我们也需要在书籍被浏览时到 PERFORMERS数据表中搜寻数据, 因此请开启数据模块, 在数据模块中再加入一个 TSimpleDataSet组件, 设置它的 DBConnection特性值为 scnnDemo, 设置它的 Name特性值为

scdsAuthor。此时数据模块类似于图 2-9。



图2-7 执行范例应用程序的画面

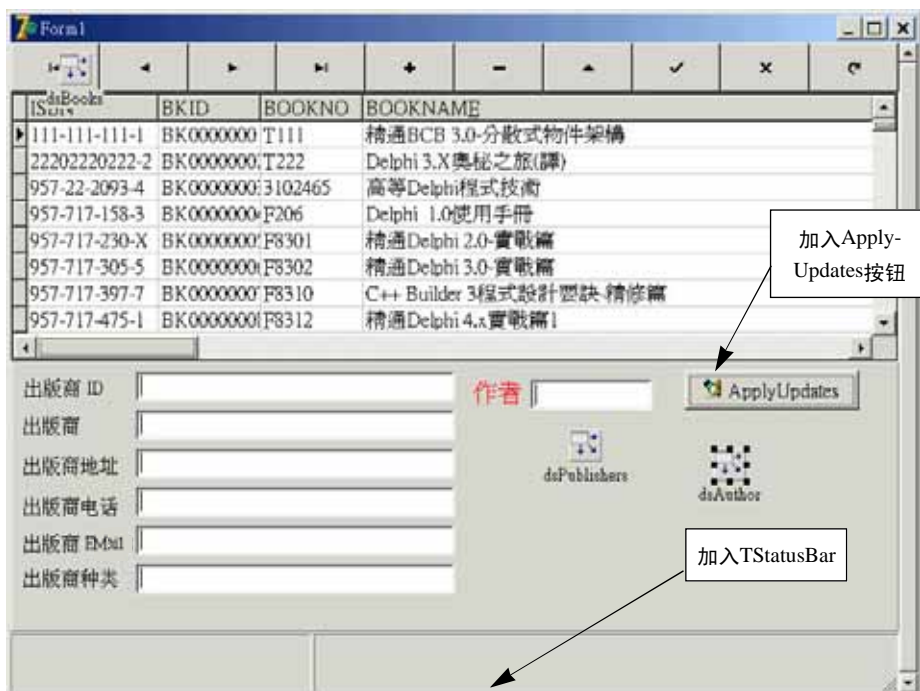


图2-8 在范例应用程序的主窗体中加入 TBitBtn 组件和 TStatusBar 组件

接着在前面已经讨论过的 AfterScroll 事件处理函数中加入一行调用 GetAuthor 的程序代码：

```
procedure TdmDemo1.scdsBooksAfterScroll
(DataSet: TDataSet;
begin
  try
    Self.scdsPublishers.Active := False;
    Self.scdsPublishers.Params.ParamByName
('VID').Value :=
      Self.scdsBooks.FieldByName('VID').Value;
    Self.scdsPublishers.Active := True;
```

```
GetAuthor;
```

```
except
  on Exception do;
end;
end;
```

而 GetAuthor 方法则使用了动态组成 SQL 语句的方式从 PERFORMERS 数据表中取得数据。下面的程序代码很简单，它从 scdsBooks 中取得书籍作者的 ID，再通过这个 ID 组成 SQL 语句从 PERFORMERS 数据表中取得正确的数据。

```
procedure TdmDemo1.GetAuthor;
begin
  Self.scdsAuthor.Active := False;
  Self.scdsAuthor.DataSet.CommandText :=
    'select * from PERFORMERS where AID = ' + '''' +
      Self.scdsBooks.FieldByName('AID').Value + '''';
  Self.scdsAuthor.Active := True;
end;
```

现在再执行范例应用程序，我们便可以在主窗体中看到书籍作者的数据了。

在这里，本书只是展示程序员也可以使用动态组成 SQL 的方式来访问数据，scdsAuthors 就是一个范例。但是这里的程序代码并不是很好的，因为它的性能并不好。如果读者使用与前面 scdsPublishers 一样的动态参数方法会比较有效率，因为在 scdsAuthors 的 SQL 语句中只有作者 ID 会改变。

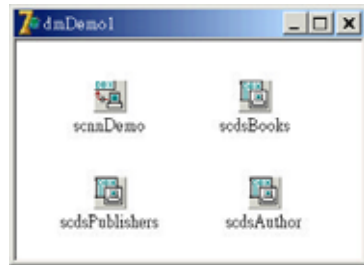


图2-9 在数据模块中加入 TSimpleDataSet 组件，设置 Name 特性值为 scdsAuthor

在前面本书说过，当用户在应用程序中修改了数据后，这些修改的数据会暂时存储在 TSimpleDataSet 的 Delta 特性值中。程序员可以通过访问 TSimpleDataSet 的组件 ChangeCount 特性值来得知目前已经被修改的数据记录数，并且可以根据这个数值来决定是否需要调用 ApplyUpdates 方法把修改的数据真正更新回后端数据源中。现在

就让我们在范例应用程序中显示目前被修改的数据记录数。

当TSimpleDataSet调用Post方法把修改的数据暂时存储在Delta特性值中后，它的ChangeCount特性值便会随着更新。因此要得知目前的修改数据记录数，我们可以在scdsBooks的AfterPost事件处理函数中显示ChangeCount。请点击数据模块中的scdsBooks，并且在它的AfterPost事件中编写如下的程序代码：

```
procedure TdmDemo1.scdsBooksAfterPost (DataSet: TDataSet);
begin
    Form1.ShowChangeCount;
end;
```

上面的程序代码调用了主窗体中的ShowChangeCount方法来显示修改的数据记录数。

接着回到主窗体中双击ApplyUpdates按钮并且在OnClick事件中编写如下的程序代码。当scdsBooks调用ApplyUpdates方法之后也调用ShowChangeCount方法再次显示当数据真正被更新回数据源之后客户端暂时内存中Delta特性值的改变状态。

```
procedure TForm1.bbbtnApplyClick (Sender: TObject);
begin
    dmDemo1.scdsBooks.ApplyUpdates (0);
    ShowChangeCount;
end;
```

最后是ShowChangeCount方法的实现，它只是访问scdsBooks的ChangeCount特性值并且在主窗体的TStatusBar组件中显示：

```
procedure TForm1.ShowChangeCount;
begin
    StatusBar1.Panels[0].Text :=
        '目前被修改的数据笔数 : ' + IntToStr
        (dmDynamicSQL.scdsBooks.ChangeCount);
end;
```

现在编译并且执行范例应用程序，图2-10是执行范例应用程序并且更新两个记录之后的画面。从TStatusBar组件中可以看到ChangeCount特性值正确地记录了目前有两个记录在客户端被修改了。

接着我们点击窗体中的ApplyUpdates按钮调用ApplyUpdates方法把数据真正更新回数据源中。图2-11显示了当ApplyUpdates方法成功执行完毕之后，客户端记录的暂时修改数据记录数也清除为0了。

到目前为止，本范例展示了如何使用dbExpress组件访问数据以及使用动态SQL在应用程序执行时动态选择应用程序需要的数据。读者现在应该能够使用dbExpress组件来进行基本的数据处理了。接下来让我们继续通过这个范例应用程序讨论重要的Data和Delta特性。

BOOKNO	BOOKNAME	VID	CATEG
F8312	精通Delphi 4.x 實戰篇1	0000000003	Delphi
F8313	精通Delphi 4.x 實戰篇2	0000000003	Delphi
F8319	C++ Builder 4程式設計進階	0000000003	BCB
F8329	實戰Delphi 5.x-分散式多層電子商務篇	0000000003	Delphi
PG20008	精通Borland C++ Builder-基礎篇	0000000007	BCB
F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇	0000000000	Delphi
FXXXX	實戰Delphi 7.x 高效率資料庫設計-dbExpress篇	0000000003	Delphi
FXXX1	實戰Delphi 7.x 專業Web Service程式設計	0000000003	Delphi

出版商 ID: 0000000003 作者: 李維 ApplyUpdates

出版商: 旗標

出版商地址: 台北市杭州南路5段131號

出版商电话: 02-2396-3257

出版商 EMail: flag@com.tw

出版商种类: BK

目前被修改的数据笔数: 2

图2-10 执行范例应用程序并且修改数据

BOOKNO	BOOKNAME	VID	CATEG
F8312	精通Delphi 4.x 實戰篇1	0000000003	Delphi
F8313	精通Delphi 4.x 實戰篇2	0000000003	Delphi
F8319	C++ Builder 4程式設計進階	0000000003	BCB
F8329	實戰Delphi 5.x-分散式多層電子商務篇	0000000003	Delphi
PG20008	精通Borland C++ Builder-基礎篇	0000000007	BCB
F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇	0000000000	Delphi
FXXXX	實戰Delphi 7.x 高效率資料庫設計-dbExpress篇	0000000003	Delphi
FXXX1	實戰Delphi 7.x 專業Web Service程式設計	0000000003	Delphi

出版商 ID: 0000000003 作者: 李維 ApplyUpdates

出版商: 旗標

出版商地址: 台北市杭州南路5段131號

出版商电话: 02-2396-3257

出版商 EMail: flag@com.tw

出版商种类: BK

目前被修改的数据笔数: 0

图2-11 范例应用程序调用ApplyUpdates方法后的画面

2.1.2 Data和Delta特性

Data和Delta分别存储了使用 dbExpress 从数据源中取得的数据以及被用户修改的数据。通过了解 Data 和 Delta 特性, 程序员能够使用一些高级的技巧来开发应用程序。先让我们观察这两个特性值在范例应用程序执行时的改变情形。

由于 Data 和 Delta 特性是 OleVariant 类型的特性, 而且包含在这个 OleVariant 数值中的数据结构是 Delphi 的 DataSnap 技术, 因此要观察这两个特性的改变, 必须使用 Delphi 的 TSimpleDataSet/ TClientDataSet 组件。我们可以在用户修改了 scdsBooks 的数据时把 scdsBooks 的 Data 和 Delta 特性值指定给另外一个 TSimpleDataSet/ TClientDataSet 的 Data 特性, 再让这个 TSimpleDataSet/ TClientDataSet 组件通过 TDataSource 连接到 TDBGrid 等数据感知组件, 这样就可以观察到 scdsBooks 的 Data 和 Delta 特性值的变化情形。

为此, 请回到范例应用程序并且开启主窗体。在主窗体中加入一个 TClientDataSet 组件, 再加入一个 TDataSource 组件, 设置它的 DataSet 特性值为刚才加入的 TClientDataSet 组件。再加入 TDBNavigator 和 TDBGrid 组件, 设置它们的 DataSource 特性值为刚才加入的 TDataSource 组件。此时主窗体如图 2-12 所示。

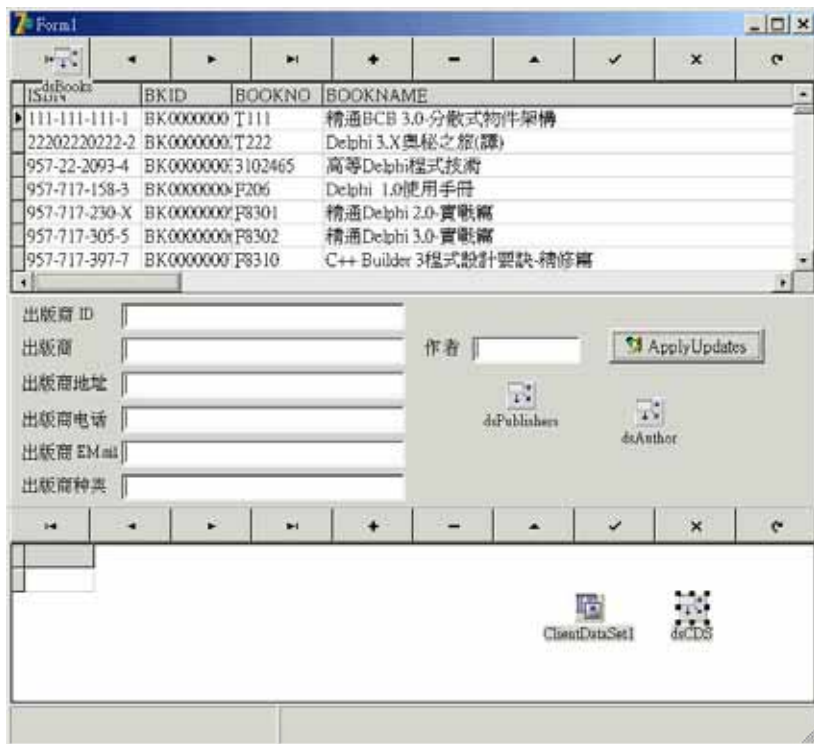


图2-12 在主窗体中加入 TClientDataSet 和其他可视化组件

现在，我们希望当用户修改了 scdsBooks 中的数据时把 scdsBooks 的 Delta 内容显示在刚才加入的 TDBGrid 中。因此我们可以在 scdsBooks 的 AfterScroll 事件中加入 ShowDelta 的过程调用代码：

```
procedure TdmDynamicSQL.scdsBooksAfterPost (DataSet: TDataSet;  
begin  
    // Self.scdsBooks.ApplyUpdate@s;  
    Form1.ShowChangeCount;  
    Form1.ShowDelta;  
end;
```

而 ShowDelta 过程关键的地方则是把 scdsBooks 的 Delta 特性值指定给刚才加入的 TClientDataSet 组件的 Data 特性值。如此一来，连接到 TClientDataSet 的 TDBGrid 便能够显示用户修改了哪些数据。

```
procedure TForm1.ShowDelta;  
begin  
    cdsDelta.Data := dmDynamicSQL.scdsBooks.Delta;  
end;
```

现在执行范例应用程序，并且试着修改 scdsBooks 中的任何数据，那么当你 Post 修改的数据之后，就会发现被修改的数据显示在主窗体下方的 TDBGrid 中。例如，图 2-13 是范例应用程序修改 FXXX1 这个记录的书名一栏的数值，在 Post 之后，请注意在下方的 TDBGrid 中显示了刚才笔者修改了 BOOKNAME 字段的数值。

在这里我们可以观察到 Delta 中的数值似乎是为每一次修改的数据保留两个记录。其中的第一个是未被更改的原始数据，而第二个则是被修改后的数据。但是在这个被修改后的记录中只有被修改的字段才会保存数值，其他没有被修改的字段则是空白的。这个现象就是 DataSnap 的运作原理，在稍后的章节中会详细说明 DataSnap 的运作方式。

既然我们已经能够通过使用 TClientDataSet 组件来巧妙地观察用户修改的数据，那么我们也可以使用相同的方式来观察 TClientDataSet 中当数据被修改后存储在 Data 中的特性值是否有任何的变化。

现在再回到主窗体中，再加入另外一组 TClientDataSet、TDBNavigator 和 TDBGrid 组件，就像刚才加入的一样。并且在主窗体的下方使用 TPageControl 组件，使用两个选项卡分别显示 Data 和 Delta 特性值。此时主窗体类似于图 2-14。

接着再修改 scdsBooks 的 AfterScroll 事件程序代码，如下所示：

```
procedure TdmDynamicSQL.scdsBooksAfterPost (DataSet: TDataSet;  
begin  
    // Self.scdsBooks.ApplyUpdate@s;  
    Form1.ShowChangeCount;  
    Form1.ShowData;
```

```
Form1.ShowDelta;
end;
```

上面的程序代码除了调用 ShowDelta之外，也调用了 ShowData来显示 scdsBooks 的 Data 特性值。而 ShowData 方法的实现程序代码则如下：

```
procedure TForm1.ShowData;
begin
    cdsData.Data := dmDynamicSQL.scdsBooks.Data;
end;
```

ShowData 也和 ShowDelta 一样，只是它把 scdsBooks 的 Data 特性值指定给新加入的 TClientDataSet 组件。

现在再次执行范例应用程序，修改任何数据，Post 修改的数据，那么就可以在方方的两个选项卡中看到 Data 和 Delta 特性值。例如，图 2-15 便是执行范例应用程序并且将原先的“实战 Delphi 7.x-Web Service 专业程序设计”修改为“实战 Delphi 7.x-Web Service 专业程序设计之二”后的画面。

The screenshot shows a Delphi application window titled 'Form1'. It contains a data table with the following data:

BKID	BOOKNO	BOOKNAME	VID
BK0000000	F8313	精通Delphi 4.x 實戰篇2	00000000
BK0000001	F8319	C++ Builder 4程式設計進階	00000000
BK0000001	F8329	實戰Delphi 5.x-分散式多層電子商務篇	00000000
BK0000001	PG20008	精通Borland C++ Builder-基礎篇	00000000
BK0000001	F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇	00000000
BK0000001	FXXXX	實戰Delphi 7.x高效率資料庫設計-dbExpress篇之一	00000000
BK0000001	FXXXX1	實戰Delphi 7.x-專業Web Service程式設計	00000000

Below the table is a form with the following fields:

- 出版商 ID: 000000003
- 出版商: 億模
- 出版商地址: 台北市杭州南路5段131號
- 出版商电话: 02-2396-3257
- 出版商 EMail: flag@com.tw
- 出版商种类: BK
- 作者: 李維
- Apply Updates button

At the bottom, there is another table showing the modified data:

ISBN	BKID	BOOKNO	BOOKNAME
95961-1755-1	BK0000001	FXXXX	實戰Delphi 7.x高效率資料庫設計-dbExpress篇
			實戰Delphi 7.x高效率資料庫設計-dbExpress篇之一

At the very bottom, it says '目前被修改的数据笔数: 1'.

图2-13 范例应用程序显示了用户修改的数据

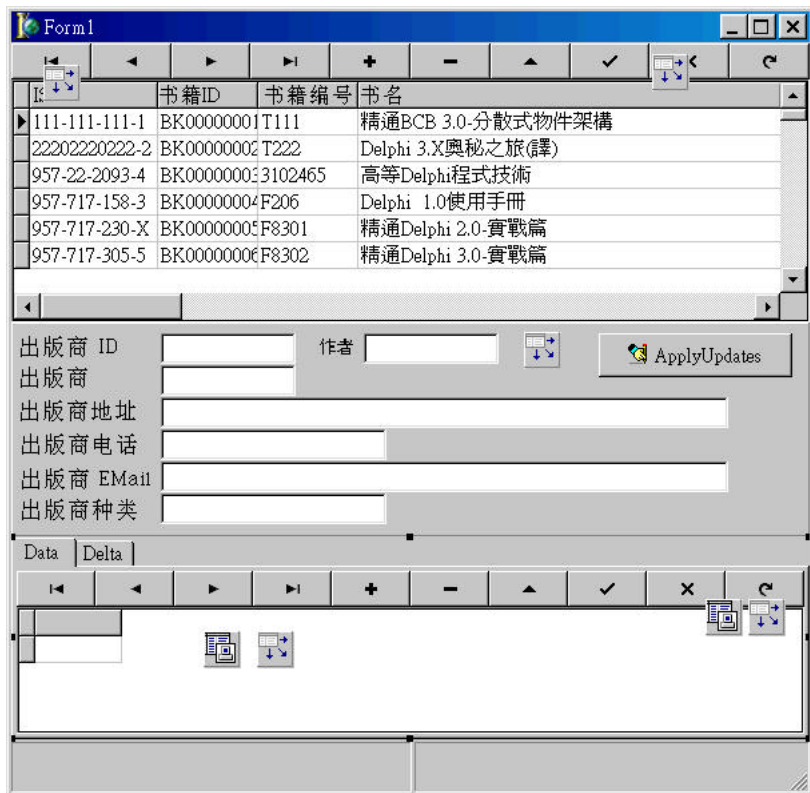


图2-14 在主窗体中再加入显示Data的数据感知组件

请注意，当Post修改的数据之后，原先存储在 `scdsBooks` 的Data特性值中的原始数据也会跟着变成新的数值。

到此为止，本范例已经介绍了如何使用 `dbExpress` 组件来访问数据并且把数据更新回数据源中。但是在这个范例的主窗体中，`ApplyUpdates` 方法只能够把数据真正更新回 `BOOKS` 数据表中。如果我们希望用户能够同时更新 `BOOKS` 和 `PUBLISHERS` 这两个数据表的数据，那么该如何做呢？下一小节会说明如何使用 `dbExpress` 更新多个数据表的数据。

2.1.3 修改数据——多个数据表

在前面的范例应用程序中，主窗体同时显示了三个数据表的数据：书籍、出版商和作者。但是在前面 `ApplyUpdates` 按钮的 `OnClick` 事件处理函数只调用了 `scdsBooks` 的 `ApplyUpdates` 方法，因此即使用户同时修改了书籍和出版商的数据并且点击 `ApplyUpdates` 按钮，那么也只有书籍的数据会更新回 `BOOKS` 数据表中，出版商的数据并不会更新回 `PUBLISHERS`，因为范例应用程序并没有调用 `scdsPublishers` 的

ApplyUpdates方法。那么范例应用程序如何同时将书籍和出版商的数据更新回这两个数据表呢？

The screenshot shows a Delphi application window titled 'Form1'. It contains two data grids and a form for publisher information.

Top Grid Data:

BKID	BOOKNO	BOOKNAME	VID
BK0000001	F8319	C++ Builder 4程式設計進階	00000000
BK0000001	F8329	實戰Delphi 5.x-分散式多層電子商務篇	00000000
BK0000001	PG20008	精通Borland C++ Builder-基礎篇	00000000
BK0000001	F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇	00000000
BK0000001	FXXXX	實戰Delphi 7.x高效率資料庫設計-dbExpress篇之二	00000000
BK0000001	FXXXX1	實戰Delphi 7.x-專業Web Service程式設計	00000000

Publisher Form Data:

出版商 ID: 000000003
 出版商: 旗標
 出版商地址: 台北市杭州南路5段131號
 出版商电话: 02-2396-3257
 出版商 EMail: flag@com.tw
 出版商种类: BK

Bottom Grid Data:

ISBN	BKID	BOOKNO	BOOKNAME
957-9625-81-6	BK0000001	PG20008	精通Borland C++ Builder-基礎篇
957-717-657-7	BK0000001	F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇
xxx-xxx-xx-xx1	BK0000001	FXXXX1	實戰Delphi 7.x-專業Web Service程式設計
xxx-xxx-xx-xx	BK0000001	FXXXX	實戰Delphi 7.x高效率資料庫設計-dbExpress篇之二

目前被修改的数据笔数: 1

图2-15 范例应用程序显示scdsBooks的Data特性值内容

也许你会认为很简单，只要在 ApplyUpdates按钮的OnClick事件处理函数中调用 scdsPublishers的ApplyUpdates方法即可。但是，如果只是这么做的话，那么可能会发生问题，因为每一个 ApplyUpdates方法会产生一个独立的数据库事务（Transaction），因此当scdsBooks和scdsPublishers分别调用ApplyUpdates方法时，便会激活两个独立的事务。因此如果用户同时更新了书籍和出版商的数据并且更新回数据表中，那么就有可能书籍数据更新成功，而出版商数据更新失败。因此如果我们希望在更新这两个数据表的数据时一定要同时成功，否则在任何一个数据表失败时必须恢复到进行更新动作之前的状态，那么这两个更新操作就必须存在于同一个数据库事务中。数据库事务可以保证在同一个事务的范围中如果发生错误就把数据恢复

到原先的状态。

数据库事务会在第4章中详细说明。

在这个范例中我们希望同时更新数据，如果任何数据表发生错误，就必须恢复数据的状态。因此我们必须把这两个数据表更新动作包含在同一个数据库事务中。

TSQLConnection组件的StartTransaction方法可以激活一个独立的数据库事务，因此我们可以先调用TSQLConnection的StartTransaction方法，再更新数据表的数据，如果数据表都成功地更新，就可以调用TSQLConnection的Commit方法保证同时更新成功。如果任何数据表发生错误，可以调用TSQLConnection的Rollback方法恢复数据。

TSQLConnection的StartTransaction方法原型如下：

```
procedure StartTransaction (TransDesc: TTransactionDesc
```

它接受一个描述事务内容的参数TransDesc。TransDesc的类型是TTransactionDesc，在TTransactionDesc中用户可以指定特定的事务ID以及事务的级别。TSQLConnection之所以需要这个参数是因为dbExpress支持嵌套事务，这是比BDE先进的地方，在第4章中会继续讨论dbExpress的事务管理功能。

了解了如何把更新数据表的操作包含在事务范围中后，我们就可以修改主窗体中ApplyUpdates按钮的OnClick事件处理函数：

```
procedure TForm1.bbtnApplyClick (Sender: TObject;
var
    aTD: TTransactionDesc;
begin
    if (not dmDemo1.scnnDemo.InTransaction) then
begin
        aTD.TransactionID := 1;
        aTD.IsolationLevel := xilREADCOMMITTED;
        dmDemo1.scnnDemo.StartTransaction (aTD);
    try
        if (dmDemo1.scdsBooks.ChangeCount) > then
            dmDemo1.scdsBooks.ApplyUpdates (0);
        if (dmDemo1.scdsPublishers.ChangeCount) > then
            dmDemo1.scdsPublishers.ApplyUpdates (0);
        dmDemo1.scnnDemo.Commit (aTD);
    except
        on e: Exception do
begin
            ShowMessage (e.message);
            dmDemo1.scnnDemo.Rollback (aTD);
end;
```

```

    end;
end;
ShowChangeCount;
end;

```

上面的程序代码首先通过 TSQLConnection 的 InTransaction 特性值来判断是否还有未结束的事务。如果没有的话，就设置一个 ID 为 1 而事务级别为 xilREADCOMMITTED 的 TTransactionDesc 对象，并且调用 TSQLConnection 的 StartTransaction 方法并且传入此对象。接着，程序判断 scdsBooks 中是否有任何被修改过的数据，如果有，就调用它的 ApplyUpdates 方法。我们也同时对 scdsPublishers 进行一样的动作。最后，如果两个数据表都成功地更新，就调用 TSQLConnection 的 Commit 方法确保更新操作的完成。当然如果发生任何错误，程序执行权便会转到 except 部分。在这里程序代码先显示发生的错误消息，再调用 TSQLConnection 的 Rollback 方法将数据恢复到进行更新操作之前的状态。由于 TTransactionDesc 是在 DBXpress 程序单元中定义的，因此读者要记得在 uses 子句中加入 DBXpress。

最后，再修改 ShowChangeCount 让它也在 TStatusBar 中显示用户对出版商修改的数据记录数。

```

procedure TForm1.ShowChangeCount;
begin
    StatusBar1.Panels[0].Text :=
        '目前被修改的数据笔数 : ' + IntToStr(dmDemo1.scdsBooks.ChangeCount);
    StatusBar1.Panels[1].Text :=
        '出版商被修改的数据笔数 : ' + IntToStr
        (dmDemo1.scdsPublishers.ChangeCount);
end;

```

现在再执行范例应用程序，并且同时修改书籍和出版商的数据。此时范例应用程序就应该看起来如图 2-16 所示。请注意下方的 TStatusBar 显示了这两个数据表目前被修改的记录数。此时，如果再点击主窗体中的 ApplyUpdates 按钮，那么数据应该会成功地更新回这两个数据表中。如果观察 TSQLMonitor 组件中的追踪消息，便可以看到如下的程序代码，这证明这两个更新操作是发生在同一个数据库事务中的。

```

INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
update BOOKS set
    BOOKNAME = ?
where
    ISBN = ? and
    BKID = ? and
    BOOKNO = ? and
    BOOKNAME = ? and
    VID = ? and

```

```

CATEGORY = ? and
CLEVEL = ? and
AID = ?
...
update PUBLISHERS set
  ADDRESS = ?
where
  VID = ? and
  NAME = ? and
  ADDRESS = ? and
  PHONE = ? and
  EMAIL = ? and
  PKIND = ?
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_retaining

```

Form1

BKID	BOOKNO	BOOKNAME	VID
BK0000001	F8319	C++ Builder 4程式設計進階	00000000
BK0000001	F8329	實戰Delphi 5.x-分散式多層電子商務篇	00000000
BK0000001	PG20008	精通Borland C++ Builder-基礎篇	00000000
BK0000001	F8331	實戰Delphi 5.x 專業ADO/MTS/COM+程式設計篇	00000000
BK0000001	FXXXX	實戰Delphi 7.x-專業Web Service程式設計	00000000
BK0000001	FXXXX1	實戰Delphi 7.x-專業Web Service程式設計	00000000

出版商 ID: 000000003
 出版商: 旗標
 出版商地址: 台北市杭州南路5段131號之一
 出版商电话: 02-2396-3257
 出版商 EMail: flag@com.tw
 出版商种类: BK

作者: 李維 Apply Updates

Data Delta

ISBN	BKID	BOOKNO	BOOKNAME
BK0000001	FXXXX		實戰Delphi 7.x-專業Web Service程式設計
			實戰Delphi 7.x-專業Web Service程式設計

目前被修改的数据笔数: 1 出版商被修改的数据笔数: 1

图2-16 在范例应用程序中同时修改 BOOKS和PUBLISHERS这两个数据表的数据

现在的范例应用程序已经很完整了，它使用了多种经常用来开发数据库应用程序的技巧，也证明了前面讨论的内容。读者可以继续修改这个范例，例如如果用户也修改了作者信息，那么该如何把作者的数据更新回 PERFORMERS 数据表呢？

2.1.4 控制数据访问记录数——PacketRecords特性

TSimpleDataSet的PacketRecords特性已经在前面介绍过了，这个特性对于整个dbExpress程序设计都有重大的影响，因此读者必须切实地了解并且掌握这个特性。PacketRecords的默认值是-1，这代表当TSimpleDataSet开启后，它会尽可能地把数据源中的数据一次读到客户端。对于拥有少量数据的数据源来说，这可能是很好的选择；但是对于拥有大量数据的数据源来说，这却是不好的设置。在一般的情形中，程序员可以将TSimpleDataSet的PacketRecords特性值调整为10到100之间，一次只访问应用程序需要的数据。这样可以让应用程序的反应时间良好，也可以节省客户端的资源。请记住一定要在真正需要一次访问所有数据时才设置 PacketRecords为-1。

现在让我们看一个小范例来说明 PacketRecords特性。首先在Delphi中建立一个新的应用程序，再建立一个新的数据模块，然后在数据模块中放入一个连接到范例数据库 D7Books的 TSQLConnection，放入一个 TSimpleDataSet组件，设置它的 DataSet\ CommandText为select * from PERFTEST。在PERFTEST数据表中拥有1000个记录，此时数据模块如图2-17所示。

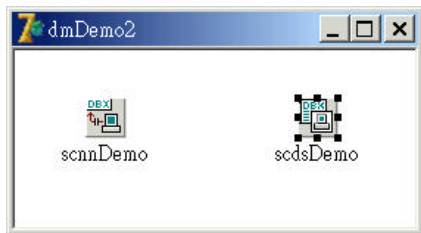


图2-17 范例应用程序的数据模块

在主窗体中加入 TDataSource、TDBNavigator、TDBGrid、TStatusBar以及一个 TButton、TSpinEdit和一个 TListBox 组件（见图2-18）。将数据感知组件设置为连接到数据模块中的 scdsDemo。

在主窗体的“设置 PacketRecords”按钮的OnClick事件处理函数中，会根据用户在TSpinEdit中设置的数值来设置 scdsDemo的PacketRecords特性值。下面就是这个按钮的OnClick事件程序代码：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    dmDemo.scdsDemo.Active := False;
    LogStartTime;
    dmDemo.scdsDemo.PacketRecords := sedtPacketRecords.Value;
    dmDemo.scdsDemo.Active := True;
    LogEndTime;
    LogMsg(sedtPacketRecords.Value, GetRunTime);
end;
```

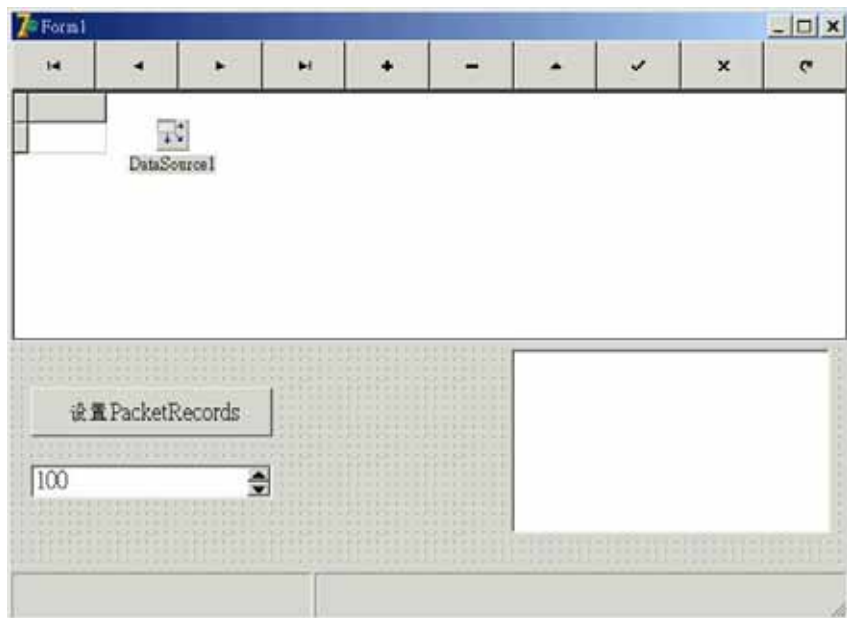


图2-18 范例应用程序的主窗体

上面的程序代码先关闭 `scdsDemo` 组件，再设置它的 `PacketRecords` 特性值，最后再开启 `scdsDemo`，并且把这整个过程花费的时间显示在主窗体的 `TListBox` 中。

此外在范例应用程序执行时，我们也想要知道目前在客户端应用程序中 `dbExpress` 到底从数据源取得了多少个记录。为此，我们可以访问 `TSimpleDataSet` 组件的 `RecordCount` 特性值来得知目前在 `Data` 特性值中的记录数。

除了知道如何访问目前客户端的记录数之外，我们也必须决定什么时候比较适合访问记录数的信息。其中第一个时机便是当 `TSimpleDataSet` 被开启的时候，那就是它的 `AfterOpen` 事件。另外一个时机便是当 `TSimpleDataSet` 需要从数据源访问下一个数据封包时，这便是它的 `AfterGetRecords` 事件。因此我们只需要在这个两个事件中显示 `TSimpleDataSet` 的 `RecordCount` 特性值就可以知道目前在客户端被访问的记录数。下面的程序代码就是本范例应用程序在这两个地方显示 `RecordCount` 特性值的程序代码：

```
procedure TdmDemo.scdsDemoAfterOpen (DataSet: TDataSet);
begin
    Form1.ShowRecordCount (scdsDemo.RecordCount);
end;

procedure TdmDemo.scdsDemoAfterGetRecords (Sender: TObject;
    var OwnerData: OleVariant);
begin
```

```

if (scdsDemo.Active) then
    Form1.ShowRecordCount (scdsDemo.RecordCount) ;
end;

```

现在编译并且执行范例应用程序。图 2-19便是将PacketRecords设置为 10时开启 TSimpleDataSet的画面。在主窗体下方的 TStatusBar3中可以看到目前客户端果然只访问了10个记录。

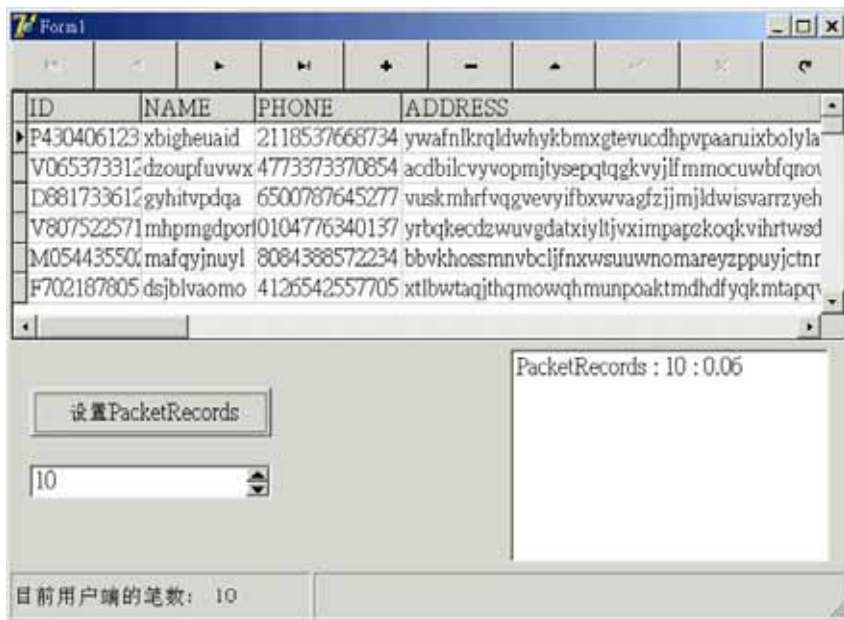


图2-19 将PacketRecords设置为10时范例应用程序的执行画面

现在，如果我们点击 TDBNavigator不断地向下浏览数据，便会发现当浏览第 11 个记录时， TSimpleDataSet会再从数据源自动取得下一个数据封包，此时 TStatusBar 也会显示记录数已经成为 20了（见图 2-20）。所以证明了 TSimpleDataSet每次以 PacketRecords指定的数目从数据源读取数据。

图2-21则是在PacketRecords为100时从数据源读取数据，不管 PacketRecords值为何， TSimpleDataSet的执行行为都是一样的，但是我们发现范例应用程序使用 100作为PacketRecords值似乎比10运行得快，读者可以自行试着调整 PacketRecords值，看看不同的PacketRecords值会有什么执行结果。

这个范例显示了 TSimpleDataSet的PacketRecords可以控制一次从数据源中取得的记录数，让程序员能够控制客户端应用程序需要的数据。另外，我们也发现不同的 PacketRecords特性值会使应用程序产生不同的反应时间，程序员可以根据自己的情况来设置PacketRecords特性值，一般来说如果 PacketRecords特性值不是设置成 - 1，那么10到1000之间是比较好的设置。当然如果使用客户端的人数多，那么程序员可

能需要将PacketRecords特性值减少为10到100之间。

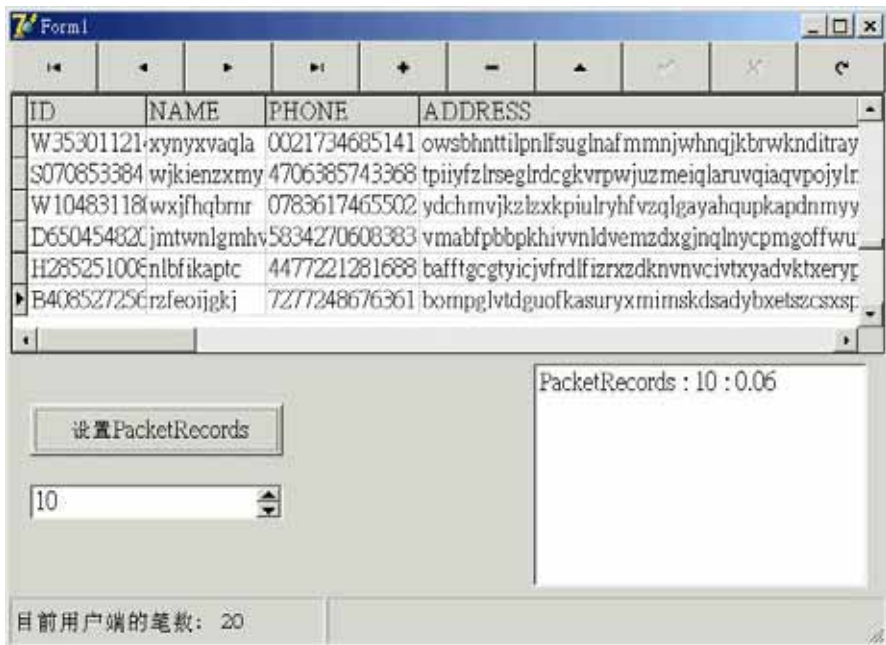


图2-20 范例应用程序会自动读取下一个数据封包

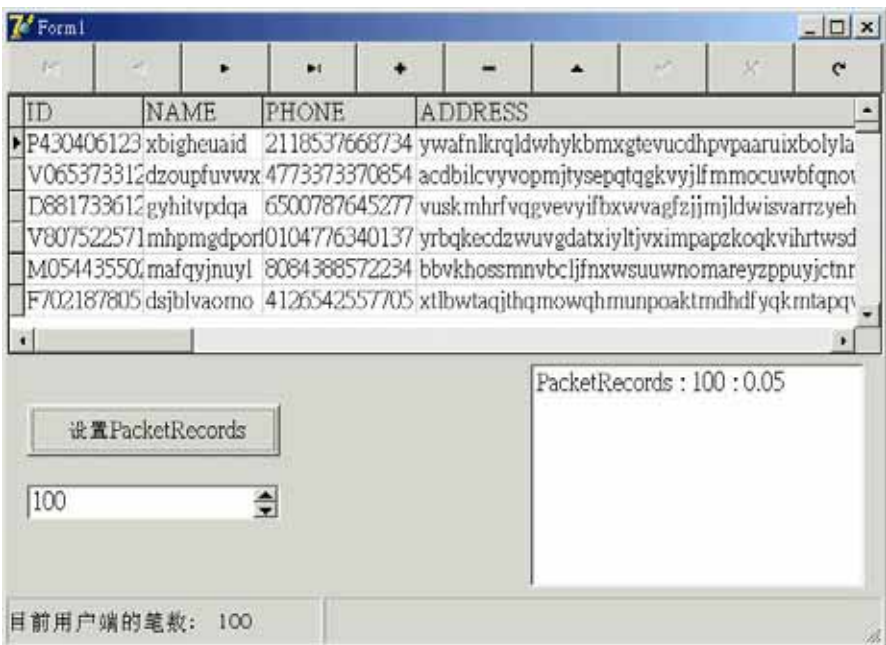


图2-21 范例应用程序使用100作为PacketRecords值时的执行画面

这本高效率数据库程序设计书其实在 Delphi 6时就已经开始编写了，不过笔者到了 Delphi 7才推出本书。在笔者把 Delphi 6的程序更改成 Delphi 7时发现 Delphi 7的dbExpress的性能又比Delphi 6时又好很多，许多执行时间比 Delphi 6时又快了10%~30%，实在令人非常惊讶且兴奋，看来 Borland仍然在不断地改善dbExpress的性能，也许在读者阅读本书时 dbExpress又有了进步。

2.2 DataSnap技术

Delphi 7的dbExpress是一组组件和数据访问引擎，虽然程序员在开发数据库应用程序时是使用dbExpress，但是在dbExpress的内部却使用了DataSnap技术。Delphi 7的DataSnap也就是以前Delphi中的MIDAS技术，只是Delphi 7强化了MIDAS的功能，还让MIDAS能够同时在Windows和Linux平台上执行。

DataSnap是以数据封包（Data Packet）的方式来传递数据的，当客户端应用程序使用dbExpress访问数据时，在dbExpress的内部会使用SQL语句从数据源取得结果数据集，然后把结果数据集中的数据转换为数据封包，再存储到 TSimpleDataSet或是 TClientDataSet的Data特性值中。而数据封包则是以 OleVariant的类型代表的，因此 TSimpleDataSet的Data和Delta特性值都是OleVariant类型的数值。

而当应用程序修改了数据时，这些修改的数据便存储在 TSimpleDataSet的Delta特性值中，并且当应用程序调用 ApplyUpdates方法更新数据时，Delta特性值便会传递到dbExpress的内部，再由DataSnap根据这些修改的数据自动转换为SQL语句，再执行这些SQL语句把数据更新回数据源中。图2-22便显示了这个处理流程。

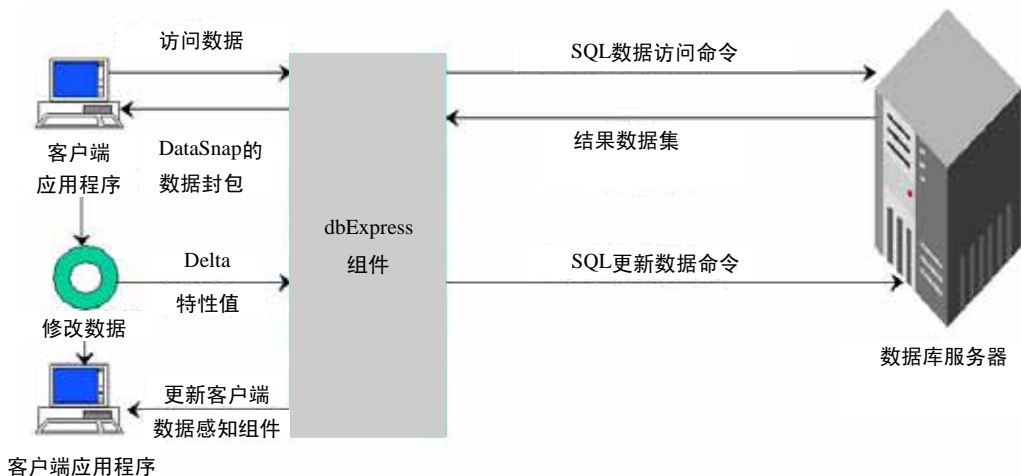


图2-22 dbExpress客户端应用程序如何从数据源取得数据和处理数据

如果客户端应用程序以分段的方式访问数据，即设置 TSimpleDataSet的Packet-Records特性值为一个正数，那么当用户在客户端浏览或是访问的数据不在目前的Data特性值中时，dbExpress和DataSnap便会自动地从后端数据源中访问下一个数据封包。TSimpleDataSet组件会自动调用 GetNextPacket访问下一个数据封包，一直到所有的数据都已经读入 TSimpleDataSet的Data特性值为止，图2-23说明了以分段访问数据的流程。

让TSimpleDataSet以分段的方式访问数据的好处是可以降低应用程序的反应时间，也可以在多人使用时降低网络的负荷。这对于多人使用的系统而言是比较好的数据访问模式。

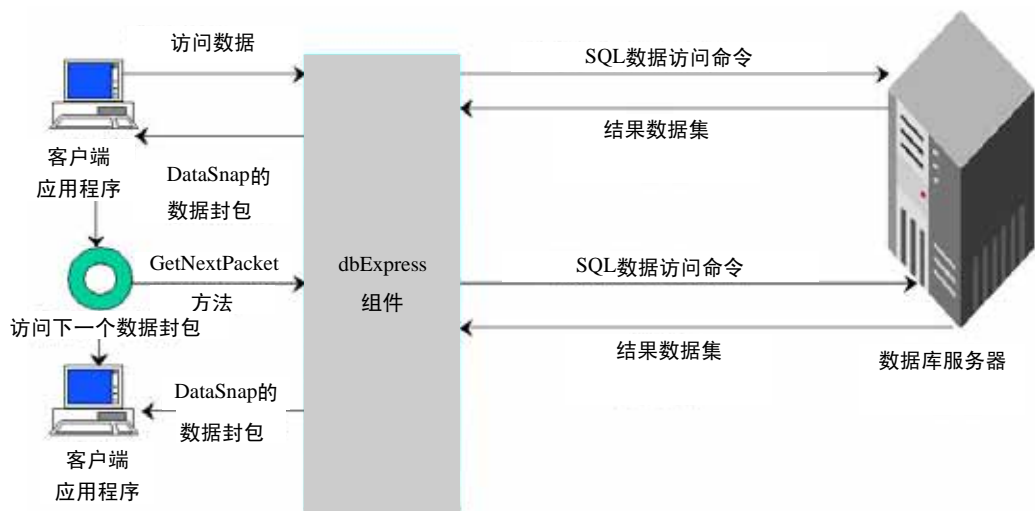


图2-23 dbExpress应用程序如何使用分段的方式访问数据

当TSimpleDataSet以分段的方式访问数据时，它也会触发 TSimpleDataSet组件中相应的事件处理函数。例如，在 GetNextPacket真正访问数据之前，它会触发 TSimpleDataSet的BeforeGetRecords事件；在访问下一数据封包之后，它又会触发 AfterGetRecords事件。刚才在前面的范例中我们也是利用 AfterGetRecords事件来显示目前在客户端的记录数。图2-24显示了分段访问数据的流程以及相应的事件。

在2.1.2小节中，本书讨论了Data和Delta特性值，并且通过范例应用程序观察到当用户修改数据之后Data和Delta特性值的变化。现在让我们以一个场景范例来说明在DataSnap中对于修改的数据的处理流程。

图2-25说明了客户端应用程序使用 dbExpress和DataSnap传递数据以及修改数据的流程。一开始，客户端应用程序要求数据源传递 100个记录，当 dbExpress和DataSnap收到要求后，就从数据源取得这 100个记录，封装成数据封包并且传递给客户端应用程序。

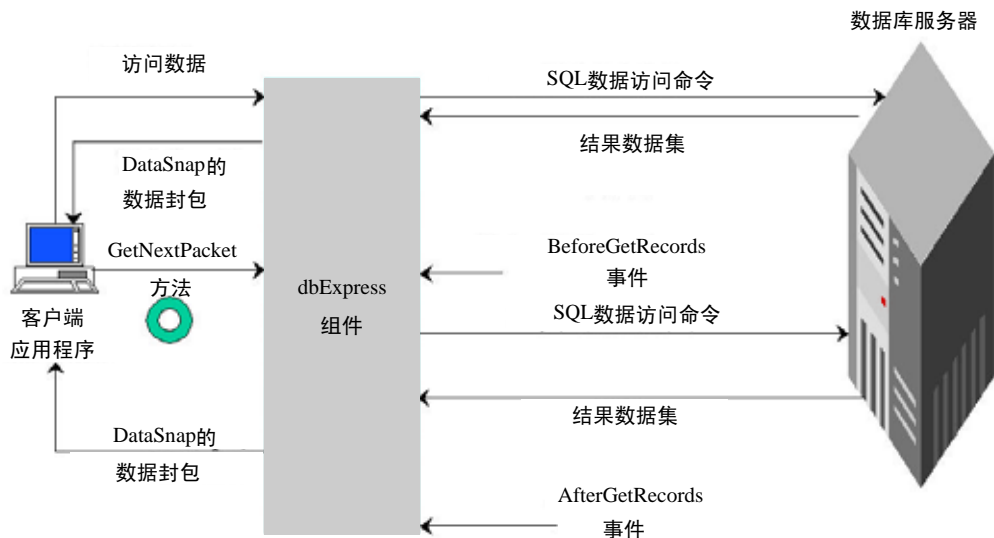


图2-24 分段访问数据时的过程以及触发的事件

客户端的用户在程序中修改数据，例如用户改变了客户端中 Joe的记录，把 Joe 的职位从 Accountant 改成 Senior Accountant，然后调用 ApplyUpdates 把这个记录封装成 Delta 数据封包返回给 dbExpress 和 DataSnap，要求更新客户端对于 Joe 这个记录的修改。

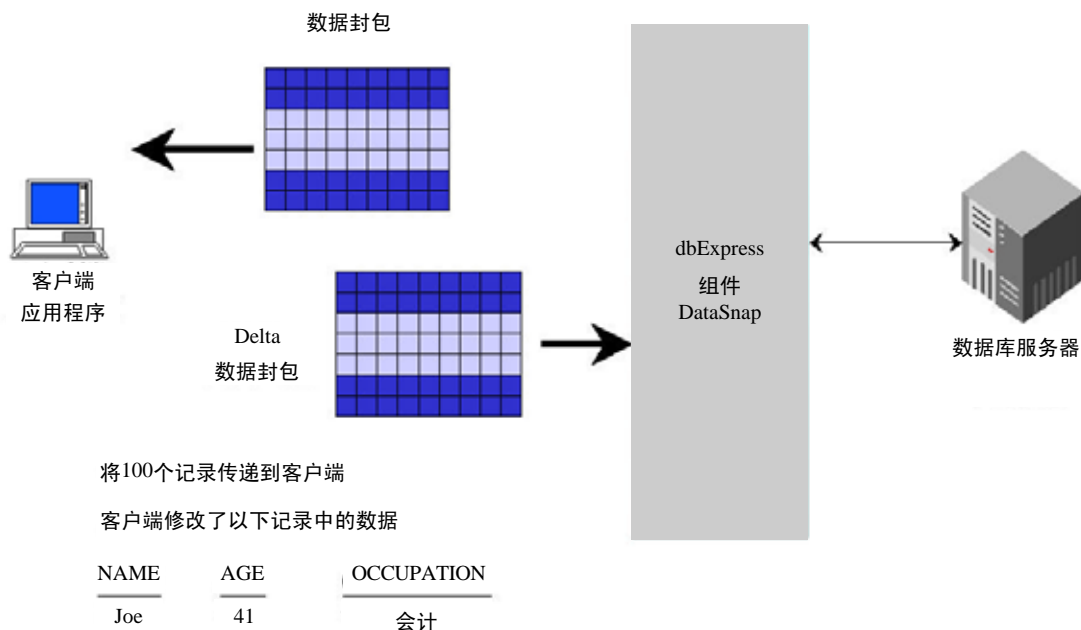


图2-25 多层应用系统传递数据的实例

当客户端封装修改数据的数据封包时，它是按照图 2-26显示的方式来封装修改数据的。首先 DataSnap 会把未修改之前的数据封装在数据封包中，然后再封装修改后的数据。请注意在封装的第二个记录中只有被改变过的字段才会有数值，此外在每一个记录中还会有一个额外的字段来说明这个记录的修改方式，例如在这个例子中 Joe 这个记录是被修改的，所以它的 OCCUPATION 字段包含 Senior Accountant 值，UPDATE TYPE 字段会拥有 Modify 值。

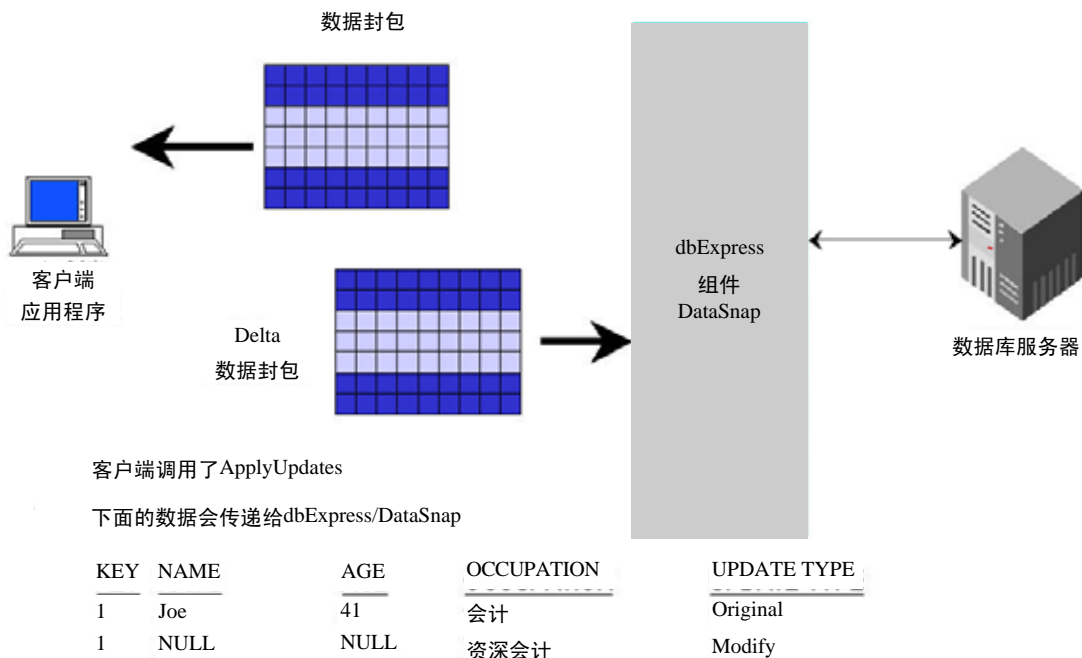


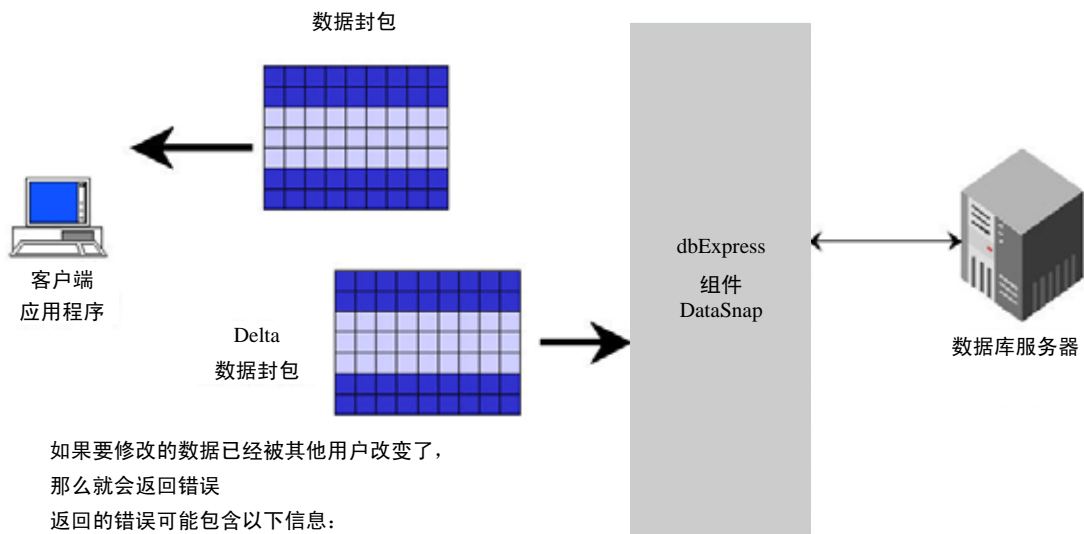
图2-26 客户端传递修改过的数据向应用程序服务器请求更新

从上面的说明中可以知道，对于在客户端修改的每一个记录而言，当这些数据要传递回数据源进行更新时，dbExpress 和 DataSnap 会封装两个记录。DataSnap 对于封装的第二个记录只记录被修改的字段，这样可以节省网络通信资源，加快传递和执行的效率。

在刚才的范例中，如果 DataSnap 发现要被更新的记录已经被其他用户改变了，例如 Joe 这个记录的 AGE 字段已经被改成 42，与原先的 41 不一样，此时 DataSnap 便会把客户端原来传送来的数据，再加上数据库中最新的这个记录传送回客户端，要求客户端应用程序或是用户决定如何处理。图 2-27 是这个数据处理流程的示意图。

当更新错误的数据到达客户端之后，客户端应用程序可以在 OnReconcileError 事件处理函数中对应用程序服务器返回的造成错误的数据进行处理。当然，客户端应用程序也可以把这些数据呈现给用户，让用户决定如何处理这些数据（见图 2-28），

在稍后的章节中会说明如何处理 dbExpress/DataSnap 更新数据发生的错误。



KEY	NAME	AGE	OCCUPATION	UPDATE TYPE
1	Joe	41	会计	Original
1	NULL	NULL	资深会计	Modify
1	Joe	42	会计	New Values

图2-27 应用程序服务器更新数据发生错误时的处理流程

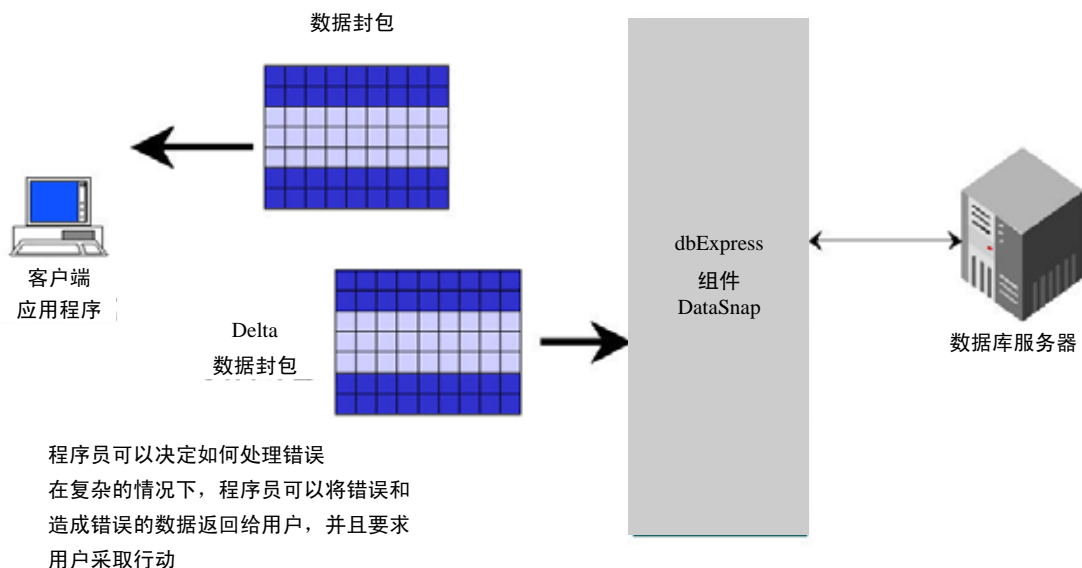


图2-28 修改数据的错误

本小节解释了 DataSnap 的基本概念和处理数据的过程。虽然在一般的 dbExpress 应用程序中程序员可能并不需要直接使用 DataSnap 技术, 因为这些工作都由 dbExpress 组件组自动帮助程序员解决了。但是在许多高级的应用中使用 DataSnap 的概念和技术却是必要的, 也可以帮助程序员解决许多困难和调整应用程序的性能。在本书稍后的章节中, 读者将会逐渐接触到 DataSnap 技术, 并且了解如何使用它发挥 dbExpress 更大的效能。

2.3 使用 TSQLDataSet 和 TSQLQuery 组件

dbExpress 的 TSQLDataSet 和 TSQLQuery 组件提供了几乎一样的功能, 只是 TSQLQuery 比较类似于 BDE 中的 TQuery 组件, 对于熟悉 BDE 的程序员来说是比较容易上手的。而 TSQLDataSet 则是一般通用的数据访问组件, 可以在许多的场合使用。

一般来说, TSQLDataSet 和 TSQLQuery 组件都是使用 SQL 语句来处理数据的。由于通过这两个组件访问的数据并不能让用户修改, 因此它们通常都是和 TClientDataSet 以及 TDataSetProvider 组件一起使用, 以便让用户可以处理和修改数据。不过对于一些属于 DDL (Data Definition Language) 的 SQL 语句来说, 这两个组件就非常适合了。例如, 建立数据库中的数据表、删除数据表或是增加索引等的工作就很适合使用这两个组件来进行。

要使用 TSQLDataSet 组件, 程序员可以设置它的 CommandType 特性来指定执行的目标是 SQL 语句或是存储过程, 就如同前面小节介绍 TSimpleDataSet 的 CommandType 特性一样。接着在 TSQLDataSet 的 CommandText 特性中下达 SQL 语句或是选择存储过程名称, 最后再设置它的 Active 特性值或是调用 ExecSQL 方法。

使用 TSQLQuery 组件和使用 TSQLDataSet 几乎一样, 只是 TSQLQuery 组件只执行 SQL 语句, 而且设置 SQL 语句的地方是在它的 SQL 特性中。简单地说, TSQLQuery 就等于一个将 CommandType 设置为 ctQuery 的 TSQLDataSet 组件。

现在就让我们使用实际的范例来说明如何使用这两个组件。

2.3.1 使用 TSQLDataSet 组件

本小节的范例将会展示如何使用 TSQLDataSet 组件来执行 DDL 语句, 在 InterBase 数据库中动态建立数据表、建立主键并且删除数据表。

步骤 1: 建立数据模块和 dbExpress 组件

在 Delphi 集成开发环境中点击 File|New|Application 建立一个新的 Delphi/Kylix 应用程序。接着点击 File|New|Data Module 建立空白的数据模块。在此数据模块中加入 TSQLConnection 组件, 连接到范例数据库 D7Books, 接着放入 3 个 TSQLDataSet 组件,

一个命名为 sdsRunDDL，另外一个命名为 sdsDropTable，最后一个命名为 sdsCreateIndex。再放入一个 TSimpleDataSet 组件，命名为 scdsQuery 并且设置它的 CommandText 特性值为 select * from Books，此时数据模块看起来如图 2-29 所示。



图2-29 范例应用程序的数据模块

在 sdsRunDDL 组件的 CommandText 特性值中加入如下的 SQL 语句。这个 SQL 语句会在数据库中建立一个名为 MYESSAYS 的数据表。

```
CREATE TABLE MYESSAYS
  EID INTEGER NOT NULL,
  ETITLE VARCHAR(60),
  MAGAZINE VARCHAR(60),
  PDATE DATE,
  CONTENTS BLOB sub_type 0 segment size 80,
  NOTES VARCHAR(100));
```



在 sdsCreateIndex 组件的 CommandText 特性值中加入如下的 SQL 语句。这个 SQL 语句会在 MYESSAYS 数据表中建立一个以 EID 字段为基础的主键。

```
ALTER TABLE MYESSAYS ADD PRIMARY KEY(EID);
```

最后一个 sdsDropTable 组件则是执行 DROP Table SQL 语句以删除由 sdsRunDDL 动态建立的数据表。

```
DROP TABLE MYESSAYS
```

步骤 2：建立范例主窗体

回到范例应用程序的主窗体，在其中放入 TDataSource 并且将它连接到数据模块中的 scdsQuery，再放入 TDBNavigator 和 TDBGrid。最后放入两个 TButton 组件，一个设置 Caption 特性值为“建立数据表”，另外一个设置 Caption 特性值为“删除数据表”。此时主窗体如图 2-30 所示。

步骤 3：实现范例应用程序

双击主窗体中的“建立数据表”按钮，并且在它的 OnClick 事件处理函数中编写如下的程序代码：

```
dmRunDDL.sdsRunDDL.ExecSQL (False) ;
dmRunDDL.sdsCreateIndex.ExecSQL (False) ;
dmRunDDL.scdsQuery.Active := False;
dmRunDDL.scdsQuery.CommandText := 'Select * from MyEssays';
dmRunDDL.scdsQuery.Active := True;
```

在上面的程序代码中，执行了 sdsRunDDL组件中的SQL语句。由于 sdsRunDDL组件的SQL语句是不返回结果数据集的 Create Table语句，因此我们必须调用 TSQLDataSet的ExecSQL方法，而不是Open方法。



图2-30 范例应用程序的主窗体

ExecSQL方法的原型如下：

```
function ExecSQL (ExecDirect: Boolean = False Integer; override;
```

ExecSQL方法用来执行不返回结果数据集的SQL语句或是属于DDL的SQL语句。它接受一个ExecDirect参数，这个参数代表 TSQLDataSet在执行SQL语句之前是否要先准备（编译）这个SQL语句。如果 ExecDirect是False就代表要先准备SQL语句，如果 ExecDirect是True则代表直接执行SQL语句而不需要先准备。如果程序员需要执行相同的SQL语句数次，那么传递 False给ExecSQL方法是比较好的。

当 sdsRunDDL执行完毕之后， MyEssays数据表应该就被建立了，因此上面的程序代码再指定 Select * from MyEssays给scdsQuery，并且开启 MyEssays数据表。如果 sdsRunDDL正确而且成功地执行，那么在范例应用程序中的 TDBGrid中应该可以看到 MyEssays这个数据表了。

接着双击主窗体中的“删除数据表”按钮，并且在它的 OnClick事件处理函数中编写如下的程序代码：

```
dmRunDDL.sdsDropTable.ExecSQL (False) ;
```

上面的程序代码调用了 sdsDropTable 的 ExecSQL 方法执行存储在 sdsDropTable 组件中的 SQL 语句，以删除 MyEssays 数据表。

步骤 4：执行范例应用程序

现在可以编译并且执行范例应用程序。图 2-31 中的画面就是执行范例应用程序的情形，在一开始时范例应用程序显示了 Books 数据表中的数据。当我们点击了“建立数据表”按钮之后，范例应用程序会动态建立 MyEssays 数据表，并且将它显示在 TDBGrid 中。



图2-31 执行范例应用程序的画面

现在，如果到数据库管理工具中检查，便可以看到类似图 2-32的画面，证明了范例应用程序果然在数据库中成功地动态建立了 MyEssays数据表。

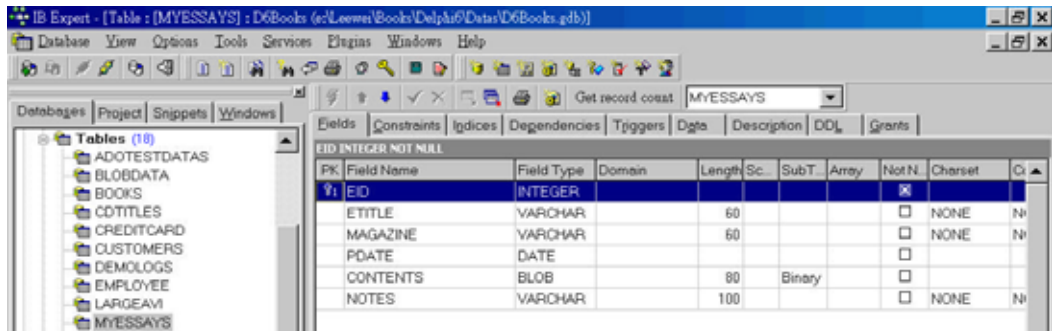


图2-32 范例应用程序在InterBase中建立了MyEssays数据表

这个范例应用程序展示了如何使用 TSQLDataSet 动态建立数据表，在随后的小节中会继续讨论如何使用 TSQLQuery 组件在动态建立的数据表中新增数据。

2.3.2 使用 TSQLQuery 组件

在上一小节中演示了如何使用 TSQLDataSet 动态建立数据表，现在让我们继续使用 TSQLQuery 来展示如何在动态建立的数据表中新增数据。

步骤 5：加入 TSQLQuery 组件

打开范例的数据模块，并且在其中加入一个 TSQLQuery 组件，命名为 sqlqData，如图 2-33 所示。

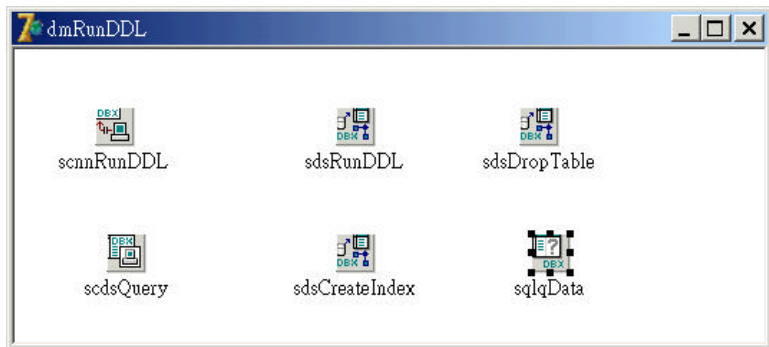


图2-33 在数据模块中加入 TSQLQuery 组件 sqlqData

设置它的 SQLConnection 特性值为数据模块中的 scnnRunDDL，并且将它的 SQL 特性值设置为：

```
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES (1, '2000年软件巨星-Kylix', 'RUN!PC 2001/3', '03/05/2001 00:00:00'), NULL
```

这个SQL语句会在MYESSAYS数据表中新增一个新记录。

步骤 6: 使用TSQLQuery组件新增数据

回到范例应用程序的主窗体，加入一个新的 TButton组件，设置它的 Caption特性值为“新增数据”，双击这个TButton组件，并且在它的 OnClick事件处理函数中编写如下的程序代码：

```
procedure TForm1.btnInsertDataClick (Sender: TObject;  
begin  
    dmRunDDL.sqlqData.ExecSQL (False) ;  
    dmRunDDL.scdsQuery.Refresh;  
end;
```

上面的程序代码调用了 sqlqData的ExecSQL方法，让它执行存储在 sqlqData组件中的SQL语句，最后再调用数据模块中的 TSimpleDataSet组件的Refresh方法从数据表中得到最新的数据。

注意到了吗？以前 BDE的TQuery在许多的情形下都无法调用 Refresh方法重新取得数据，而dbExpress的TSQLDataSet、TSQLQuery和TSQLClientDataSet组件却都可以自由地调用 Refresh，这比BDE好用得多。

现在再次执行范例应用程序，点击“建立数据表”按钮，再点击“新增数据”按钮，那么便会看到类似图 2-34的画面，TSQLDataSet和TSQLQuery组件都可以成功地执行DDL语句。



图2-34 范例应用程序动态创建数据表以及新增数据的画面

前面的范例都展示了如何使用 TSQLDataSet和TSQLQuery来执行DDL或是直接使用SQL语句来处理数据, 不过我们可以发现 TSQLDataSet和TSQLQuery一次都只能执行一个SQL语句。当程序员需要执行大量的 SQL语句(即所谓的SQL脚本)时, 由于dbExpress没有提供类似TSQLScript的组件, 因此程序员只能一次执行一个SQL语句, 相当不方便。不过在 dbExpress提供TSQLScript组件之前, 我们也可以使用TSQLDataSet和TSQLQuery来模仿TSQLScript组件, 让程序员一次能够执行多个SQL语句。下一小节就展示了这个范例。

2.3.3 执行SQL脚本

本小节要展示的范例是让 TSQLDataSet和TSQLQuery执行和2.3.1小节一样的功能, 只是本节的范例能够一次执行数个SQL语句, 而无需一次执行一个SQL语句。

步骤 1: 建立数据模块和dbExpress组件

在Delphi/Kylix集成开发环境中点击 File|New|CLX Application, 再点击 File|New|Other...菜单, 选择CLX Data Module图标, 以建立一个CXL数据模块。接着在此数据模块中加入 TSQLConnection组件并且将它连接到范例数据库 D7Books, 加入 TSimpleDataSet组件, 设置它的 DataSet\CommandText 特性值为 select * from MYESSAYS。加入 TSQLDataSet组件和一个TSQLQuery组件, 如图2-35所示。当然, 这些组件都需要连接到数据模块中的TSQLConnection组件。

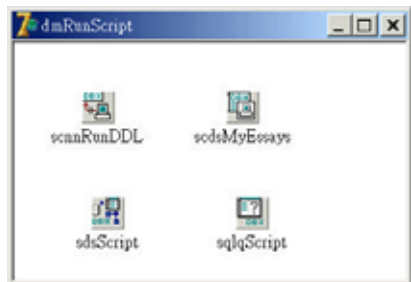


图2-35 范例应用程序的数据模块

本范例要执行的SQL脚本如下:

```
CREATE TABLE MYESSAYS
(
  EID INTEGER NOT NULL,
  ETITLE VARCHAR60),
  MAGAZINE VARCHAR60),
  PDATE DATE,
  CONTENTS BLOB sub_type 0 segment size 80,
  NOTES VARCHAR100));

INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(1, '2000年软件巨星-Kylix', 'RUN!PC 2001/3', '03/05/2001 00:00:00', NULL);
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(2, 'Windows原生开发工具的瑰宝-Delphi 7', 'RUN!PC 2001/6', '05/05/2001 00:00:00', NULL);
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(3, 'So much fun, So many possibilities', 'RUN!PC 2001/7', '06/05/2001 00:00:00', NULL);
```

```
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTVALUES
(4, '肥皂的战争与和平', 'RUN!PC 2001/8', '07/05/2001 00:00:00'), NULL
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTVALUES
(5, '.NET的SOAP', 'RUN!PC 2001/9', '08/05/2001 00:00:00'); NULL
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTVALUES
(990, '我的回忆和有趣的故事', 'Programme深度论坛', '03/01/2001 00:00:00', NULL
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTVALUES
(991, '我的回忆和有趣的故事续之一', 'Programme深度论坛', '04/01/2001 00:00:00',
NULL);
```

```
ALTER TABLE MYESSAYS ADD PRIMARY KEY(EID);
CREATE INDEX MYESSAYS_IDX1 ON MYESSAYS(ETITLE);
CREATE INDEX MYESSAYS_IDX2 ON MYESSAYS(PDATE);
```

这个SQL Script会建立MyEssays数据表，新增数个记录，再建立主键值以及数个索引对象。

步骤 2：建立主窗体

回到范例应用程序的主窗体。在主窗体中加入一个 TMemo组件以显示要执行的SQL脚本。放入 TDataSource并且连接到数据模块中的 TSimpleDataSet组件，放入 TDBNavigator和TDBGrid并且连接到TDataSource。放入一个 TOpenDialog组件以便开启SQL脚本文件。最后放入两个 TButton组件，一个加载SQL脚本文件，而另外一个负责执行加载的SQL脚本。这个范例应用程序的主窗体如图 2-36所示。

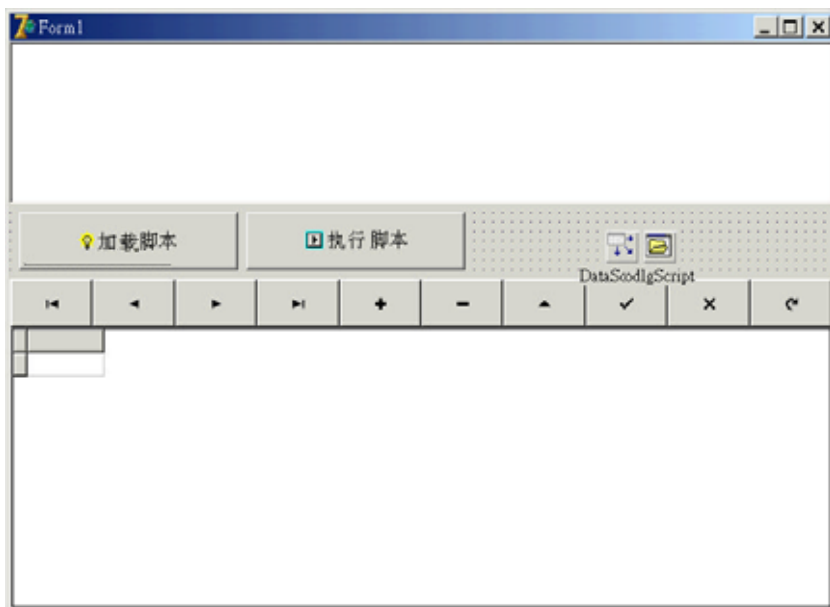


图2-36 范例应用程序的主窗体

现在我们就可以开始实现这个范例应用程序了。

步骤 3: 加入实现程序代码

首先双击主窗体中的“加载脚本”按钮，在它的 OnClick 事件处理函数中使用 TOpenDialog 组件加载 SQL 脚本文件，并且把脚本的内容显示在 TMemo 中：

```
procedure TForm1.bbtnLoadScriptClick (Sender: TObject;
begin
    if (odlgScript.Execute) then
        mmScript.Lines.LoadFromFile (odlgScript.FileName) ;
end;
```

接着双击“执行脚本”按钮，在它的 OnClick 事件处理函数中编写如下的程序代码：

```
procedure TForm1.bbtnRunScriptClick (Sender: TObject;
const
    sTAG = ';' ;
var
    sScript : String;
    sSQL : String;
    iPos : Integer;
begin
    sScript := Trim(mmScript.Lines.Text) ;
    while True do
    begin
        iPos := Pos(sTAG, sScript);
        if (iPos > 0) then
        begin
            sSQL := Copy(sScript, 1, iPos - 1) ;
            RunScript (sSQL) ;
            Delete (sScript, 1, iPos);
        end;
        if (Length (sScript) = 0) then
            break;
        end;
        dmRunScript.scdsMyEssays.Active := True;
    end;

    procedure TForm1.RunScript (const sSQL: String;
    begin
    {
        dmRunScript.sdsScript.CommandText := sSQL;
        dmRunScript.sdsScript.ExecSQL (True) ;
```

```
}  
dmRunScript.sqlqScript.SQL.Text := sSQL;  
dmRunScript.sqlqScript.ExecSQL (True);  
end;
```

上面的程序代码首先从 TMemo 中取得 SQL 脚本，然后使用 Pos 从 SQL 脚本中一一取出每个 SQL 语句，再调用 RunScript 来执行这个 SQL 语句。在 RunScript 方法中，可以使用 TSQLDataSet 或是 TSQLQuery 组件来执行传入的 SQL 语句。请注意，在 RunScript 方法中调用 TSQLDataSet 或是 TSQLQuery 组件的 ExecSQL 方法时，传入了 True 作为参数。这代表我们要 TSQLDataSet 或是 TSQLQuery 组件直接执行 SQL 语句而不是先准备 SQL 语句再执行，因为通常 SQL 脚本中的 SQL 语句只会执行一次，因此不需要先准备 SQL 语句，这样可以增加一点效率。

最后当所有的 SQL 脚本执行完毕之后，再调用数据模块中的 TSimpleDataSet 的 Refresh 方法重新从 MyEssays 数据表中取得所有的数据并且显示在主窗体的 TDBGrid 中。

步骤 4：执行范例应用程序

现在可以编译和执行这个范例应用程序了，图 2-37 显示了范例应用程序加载一个 SQL 脚本文件，动态建立 MyEssays 数据表，再新增数个记录，最后建立主键值和索引对象。图 2-38 使用数据库管理工具证明了这个范例应用程序果然成功地在数据库库中建立了各种数据库对象。

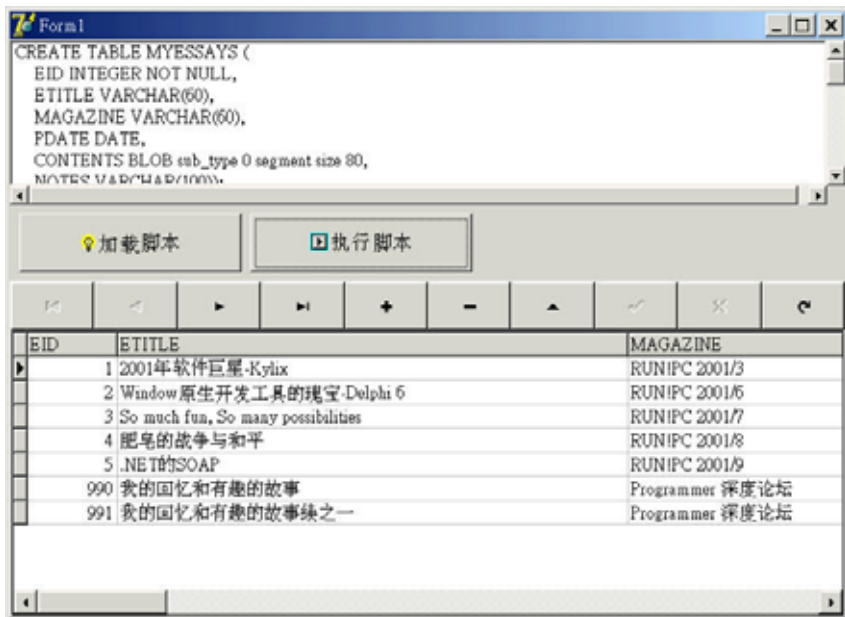


图2-37 执行范例应用程序的画面

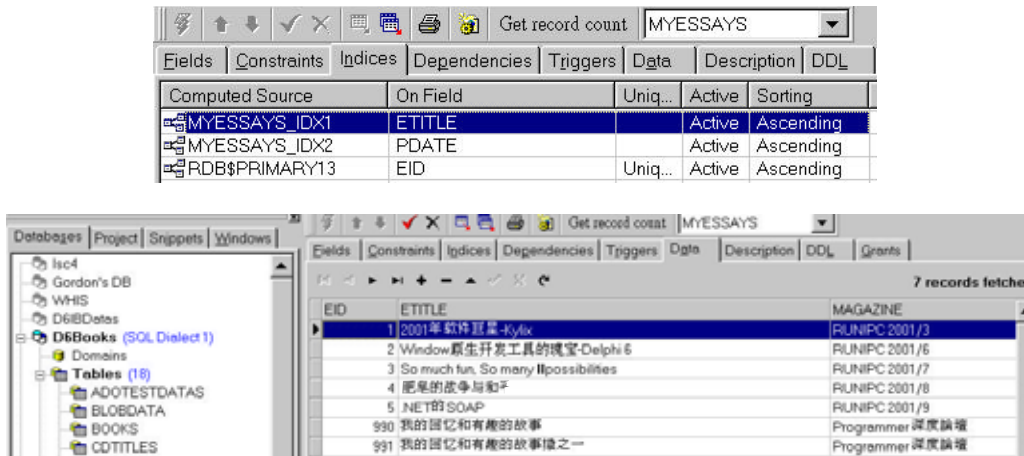


图2-38 范例应用程序建立了数据表以及索引对象

本小节讨论了如何使用 TSQLDataSet 以及 TSQLQuery 组件。虽然这两个组件都无法让用户修改它们访问的数据，但是它们可以搭配 TSimpleDataSet 组件让程序员可以开发数据库应用程序。对于一些执行 DDL 的应用程序或是新增、更新和删除大量数据的情形，使用 TSQLDataSet 和 TSQLQuery 组件比使用 TSimpleDataSet 更适当。在应用系统有这种需求时，程序员便可以使用这两个组件来弥补 TSimpleDataSet 组件不足的地方。

2.4 使用 TSQLStoredProc 组件

dbExpress 的 TSQLStoredProc 组件允许程序员用它执行后端数据源中定义的存储过程。使用 TSQLStoredProc 组件非常简单，只需要设置它的 DBConnection 特性值为一个 TSQLConnection，再设置它的 StoredProcName 特性值为欲执行的存储过程即可。如果欲执行的存储过程需要传入参数，那么只需通过 TSQLStoredProc 组件的 Params 特性值传递参数给存储过程即可。现在让我们使用一个实际的范例来展示如何使用 TSQLStoredProc 组件。

图2-39是一个在 InterBase 中实现的存储过程 RAISESALARY。这个存储过程接受两个参数，第一个是员工的 ID，第二个参数是欲加薪的百分比。RAISESALARY 可以改变 EMPLOYEE 数据表中 SALARY 字段的数值，为特定的员工加薪，现在让我们看看如何使用 TSQLStoredProc 组件来调用它。

```
CREATE PROCEDURE RAISESALARY (EID VARCHAR(10), RPERCENT FLOAT)
AS
begin
    /* Procedure Text */

```



```

Update EMPLOYEE
set
    SALARY = SALARY *(1 + :RPercent)
where
    EID = :EID;
suspend;
end

```

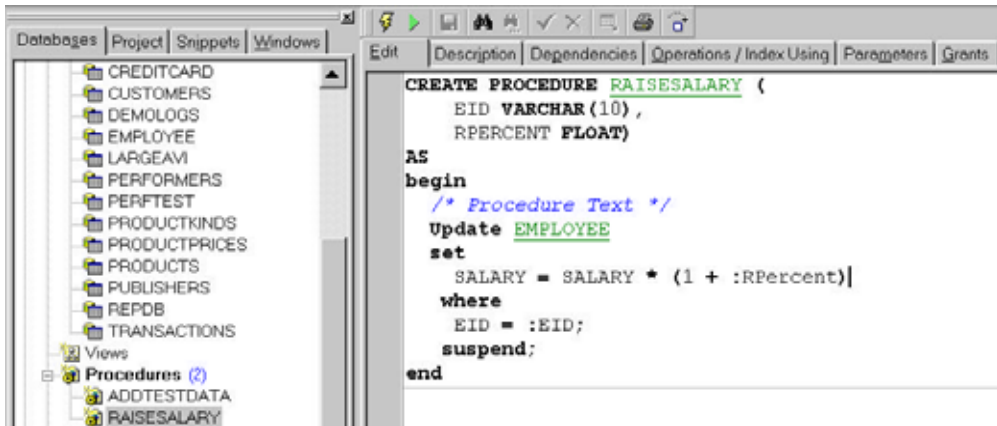


图2-39 在InterBase中定义的RAISESALARY存储过程

步骤1：建立数据模块和dbExpress组件

在Delphi/Kylix集成开发环境中点击 File|New|Application 建立一个新的 Delphi 应用程序。接着点击 File|New|Data Module 建立空白的数据模块。在数据模块中放入 TSQLConnection 组件并且连接到范例 InterBase 数据库 D7Books.GDB，设置 Connected 特性值为 True。再放入 TSimpleDataSet 组件，设置它的 DBConnection 特性值为刚才加入的 TSQLConnection，再设置它的 DataSet\CommandText 特性值为 select * from EMPLOYEE，设置 Active 特性值为 True，从 EMPLOYEE 数据表中选取所有的数据。最后在数据模块中加入一个 TSQLStoredProc 组件，设置它的 SQLConnection 特性值为刚才加入的 TSQLConnection，再点击它的 StoredProcName 特性。对象检视器便会显示目前数据库中可以调用的所有存储过程。请从其中选择 RAISESALARY 这个存储过程。此时数据模块看起来如图 2-40 所示。

步骤2：建立主窗体

在范例应用程序的主窗体中先点击 File|Use Unit... 菜单，选择使用刚才建立

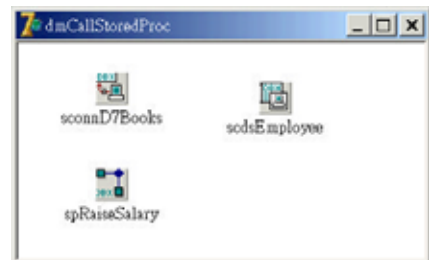


图2-40 范例应用程序的数据模块

的数据模块。接着在主窗体中加入 TDataSource 组件，设置它的 DataSet 特性值为数据模块中的 TSimpleDataSet 组件。再加入 TDBNavigator 组件和 TDBGrid 组件，设置它们的 DataSource 特性值为刚才加入的 TDataSource。最后放入一个 TButton 组件、一个 TEdit 组件和一个 TComboBox 组件。设置 TButton 组件的 Caption 特性值为“加薪”，设置 TEdit 组件的 Text 特性值为 0.05。此时主窗体看起来如图 2-41 所示。

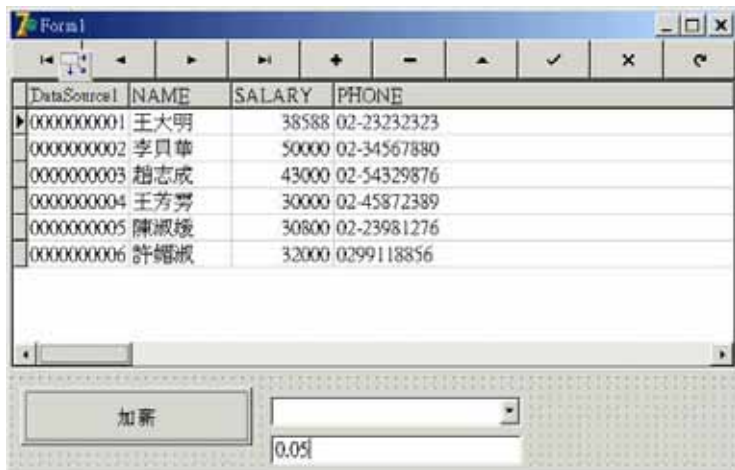


图2-41 范例应用程序的主窗体

步骤3：实现范例应用程序

现在到了实现范例应用程序的时候了。首先，在主窗体的 OnActivate 事件处理函数中取出 EMPLOYEE 数据表中所有记录 EID 字段值，填入主窗体的 TComboBox 中以便让用户能够选择要增加那一个员工的薪水。

```

procedure TForm1.FormActivate (Sender: TObject;
var
    aBK : TBookmark;
begin
    if (cbEID.Items.Count = 0) then
    begin
        aBK := dmCallStoredProc.scdsEmployee.GetBookmark;
        try
            cbEID.Items.BeginUpdate;
            dmCallStoredProc.scdsEmployee.First;
            while not dmCallStoredProc.scdsEmployee.EOF
            begin
                cbEID.Items.Add (dmCallStoredProc.scdsEmployee.FieldName ('EID').Value);
                dmCallStoredProc.scdsEmployee.Next;
            end;
        end;
    
```

```
cbEID.ItemIndex := 0;  
finally  
    cbEID.Items.EndUpdate;  
    dmCallStoredProc.scdsEmployee.GotoBookmark (aBK);  
    dmCallStoredProc.scdsEmployee.FreeBookMark (aBK);  
end;  
end;  
end;
```

上面的程序代码首先使用 TBookmark 的功能记录目前的位置，再一一取出 EID 字段值填入 TComboBox 中，最后再通过 TBookmark 回到起始的位置。

最后就是这个范例应用程序的主体了。在“加薪”按钮的 OnClick 事件处理函数中我们编写如下的程序代码：

```
procedure TForm1.btnRaiseSalaryClick (Sender: TObject;  
begin  
    dmCallStoredProc.spRaiseSalary.Params.ParamByName ('EID').Value :=  
        cbEID.Text;  
    dmCallStoredProc.spRaiseSalary.Params.ParamByName ('RPERCENT').Value :=  
        StrToFloat (edtPercent.Text);  
    dmCallStoredProc.spRaiseSalary.ExecProc;  
    dmCallStoredProc.scdsEmployee.Refresh;  
end;  
  
procedure TForm1.cbEIDChange (Sender: TObject;  
begin  
    dmCallStoredProc.scdsEmployee.Locate ('EID', cbEID.Text, [])  
end;
```

当用户点击了“加薪”按钮后，我们首先把主窗体中 TEdit 组件指定的加薪幅度通过数据模块中的 TSQLStoredProc 组件填入第二个参数中。要想填入存储过程需要的参数，可以使用 TSQLStoredProc 的 Params 特性值的 ParamByName 方法来找到特定的参数。我们也使用相同的方法把用户在主窗体中的 TComboBox 中选择的员工 ID 填入存储过程的第一个参数中。最后调用 TSQLStoredProc 组件的 Execute 方法执行存储过程，并且调用 TSQLClientDataSet 的 Refresh 方法重新显示加薪后的最新数据。

现在编译并且执行这个范例。图 2-42 中的画面就是执行范例应用程序并且为 0000000005 员工加薪的画面。从画面中可以看到我们使用 TSQLStoredProc 组件成功地调用了 RAISESALARY 存储过程并且完成了加薪的操作。

在上面的范例中，我们调用的是一个不返回数值的存储过程。那么对于会返回数值的存储过程应该如何调用呢？例如，图 2-43 中的 GetRecordCount 存储过程会计算 Books 数据表中所有书籍的总数并且返回这个数值。对于这个存储过程，我们应该如何使用 TSQLStoredProc 来调用呢？

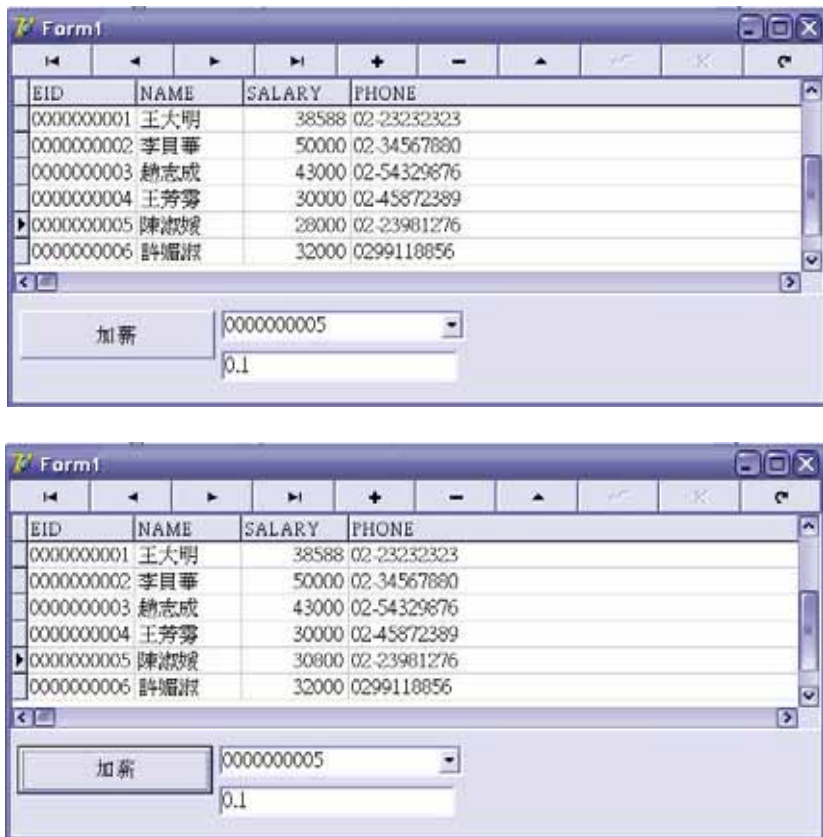


图2-42 执行范例应用程序的画面

```

Edit | Description | Dependencies | Operations / Index Using | Parameters | Grants
create procedure GetRecordCount
returns (iRecordCount integer)
as
begin
    select count(*) from Books into :iRecordCount;
    suspend;
end;

```

图2-43 在InterBase中定义的GetRecordCount存储过程

这也很简单。对于会返回数值的存储过程而言，当使用 TSQLStoredProc 组件调用时，在存储过程执行完毕之后，存储过程会把返回的数值存储在 TSQLStoredProc 组件的 Params 特性值中。程序员只需要调用 Params 的 ParamByName 方法，并且以存储过程返回的数值的名称作为 ParamByName 方法的参数即可。但是 Delphi 7 联机帮助在这个地方说错了。要想调用会返回数值的存储过程，程序员仍然需要调用 TSQLStoredProc 组件的 ExecProc 方法，而不是设置 Active 特性值为 True 或是调用 Open 方法。

下面就是调用图 2-43 中的存储过程的程序代码，在调用 ExecProc 方法之后，再调用 ParamByName 方法并且传入 IRECORDCOUNT 作为参数值即可。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    dmSP.spBookCount.ExecProc;
    Edit1.Text :=
        IntToStr(dmSP.spBookCount.Params.ParamByName('IRECORDCOUNT').Value);
end;
```

而图 2-44 就是一个范例应用程序调用 GetRecordCount 存储过程的画面，GetRecordCount 存储过程果然返回了目前 Books 数据表中的记录数，16。



图2-44 调用GetRecordCount存储过程的结果

dbExpress的 TSQLStoredProc 简化了调用存储过程的过程，程序员可以使用 TSQLStoredProc 组件来调用 dbExpress 支持的关系型数据库中的存储过程。

2.5 使用 TSQLMonitor 组件

当程序员开发应用程序时，经常会需要进行调试工作。除了调试应用程序的程序代码错误之外，也会想要观察应用程序对于后端数据源使用了什么 SQL 语句来处理或是修改数据，以便了解应用程序是否使用了正确的 SQL 语句。当然在许多情形中，能够观察应用程序对于后端数据源使用的 SQL 语句也有助于进行性能调整的工作，这对于程序员的帮助也非常大。

dbExpress 吸引人的地方是它提供了一个 TSQLMonitor 组件，这个组件使程序员可以追踪前端应用程序对于后端数据源使用的 SQL 语句。程序员不但可以通过 TSQLMonitor 组件进行调试，也可以利用它了解 dbExpress 是如何与后端数据源交互的，以便了解 dbExpress 和后端数据源的执行行为。

使用 TSQLMonitor 组件非常简单，只需要将它连接到 TSQLConnection 并且在它的事件处理函数中编写程序代码，便可以获取 dbExpress 组件和后端数据源之间互相

传递的信息。现在就让我们说明如何使用它。

步骤1: 加入TSQLMonitor组件

首先开启2.1小节建立的范例应用程序，打开数据模块，并且在数据模块中加入一个TSQLMonitor组件，如图2-45所示。



图2-45 范例应用程序的数据模块

接着设置TSQLMonitor的SQLConnection特性值为数据模块中的scnnDemo组件，再设置它的Active特性值为True以激活追踪SQL的功能。

步骤2: 修改主窗体

回到范例应用程序的主窗体，在TPageControl中加入一个新的选项卡，并且在其中放入一个TMemo组件，设置它的Name特性值为mmSQLLog，如图2-46所示。

这个TMemo组件将会显示所有由TSQLMonitor组件追踪的消息。

步骤3: 实现追踪程序代码

TSQLMonitor组件有两个事件可以让程序员用来追踪dbExpress的消息，分别是OnTrace以及OnLogTrace。OnTrace事件发生在dbExpress和后端数据源间有任何交互消息时，而OnLogTrace则会发生在交互消息已经记录进TSQLMonitor的TraceList特性中之后。要想追踪dbExpress和后端数据源之间的消息，程序员可以根据需要来决定使用哪一个事件。OnTrace以及OnLogTrace事件处理函数的原型如下：

```
type TTraceEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc;  
var LogTrace: Boolean) of object;  
property OnTrace: TTraceEvent;  
type TTraceLogEvent = procedure (Sender: TObject; CBInfo: .pSQLTRACEDesc  
of object  
property OnLogTrace: TTraceLogEvent;
```

在OnTrace以及OnLogTrace事件处理函数中，程序员可以通过访问 pSQLTRACEDesc数据结构中的pszTrace来取得dbExpress和后端数据源之间的命令。

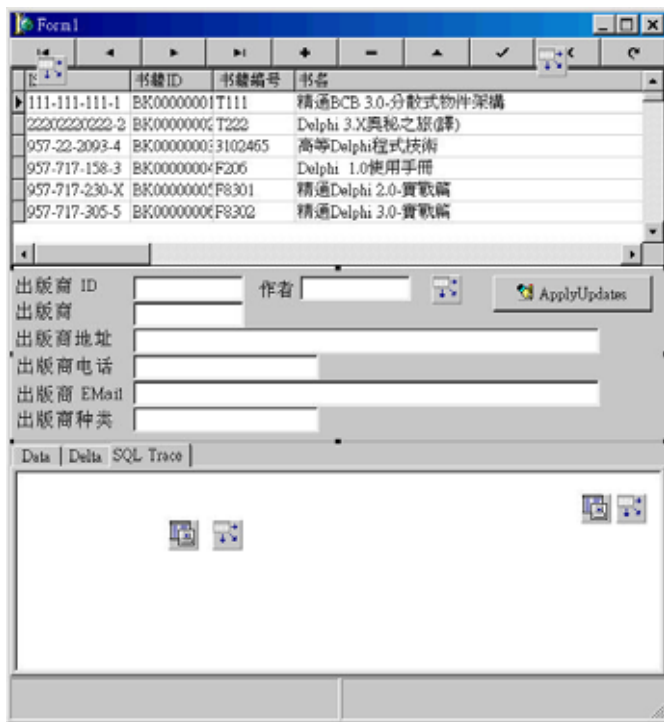


图2-46 在范例应用程序中加入一个新的选项卡和 TMemo组件

了解了TSQLMonitor的这两个事件之后，本节的范例应用程序便可以使用 OnTrace 事件来记录dbExpress的命令。请点击数据模块中的 TSQLMonitor组件，再使用对象检视器编写它的 OnTrace事件处理函数：

```
procedure TdmDynamicSQL.smSQLLogTrace (Sender: TObject;  
    CBInfo: pSQLTRACEDesvar LogTrace: Boolean;  
begin  
    Form1.mmSQLLog.Lines.Add (CBInfo.pszTrace) ;  
end;
```

在 OnTrace 事件中，我们把 dbExpress 和后端数据库之间的命令显示在主窗体的 TMemo 中。

步骤4：执行范例应用程序

现在可以编译并且执行范例应用程序了，图 2-47 是范例应用程序一开始执行时的画面。从画面中可以看到 dbExpress 下达的命令都显示在主窗体的 TMemo 组件中。

现在，先让我们清除主窗体 TMemo 组件中的消息，因为我们要开始观察 dbExpress 如何处理修改数据的工作。首先让我们随便修改一个记录并且 Post 修改的数据。图 2-48 显示了当范例应用程序 Post 修改的数据时，TSQLMonitor 组件并没有追

踪到dbExpress和后端数据源之间有任何命令，这表示在使用 Post方法进行更新时，dbExpress只是更新暂时存储在内存中的数据，也就是 TSimpleDataSet组件的Data和Delta特性值，并不会真正把修改的数据更新回数据源中。



图2-47 执行范例应用程序时，SQL命令会显示在主窗体的TMemo中

但是如果我们接着点击主窗体中的 ApplyUpdates按钮，那么范例应用程序便会在主窗体的TMemo组件中显示由dbExpress组件向后端数据源下达的命令，这些消息全被TSQLMonitor追踪到了，如图2-49所示。

我们可以通过 TSQLMonitor组件证明前面章节讨论的 DataSnap技术的执行行为，也可以观察到dbExpress是如何使用SQL语句的，以及dbExpress使用的SQL语句是不是有效率。TSQLMonitor组件使用起来非常方便，实用性却非常强，连 Borland的工程师也是使用TSQLMonitor来观察和调整dbExpress以及dbExpress引擎的执行行为。

本小节的范例说明了如何使用 TSQLMonitor组件，在稍后讨论性能的章节中本书会说明如何通过 TSQLMonitor组件来观察dbExpress应用程序的执行行为并且调整性能。

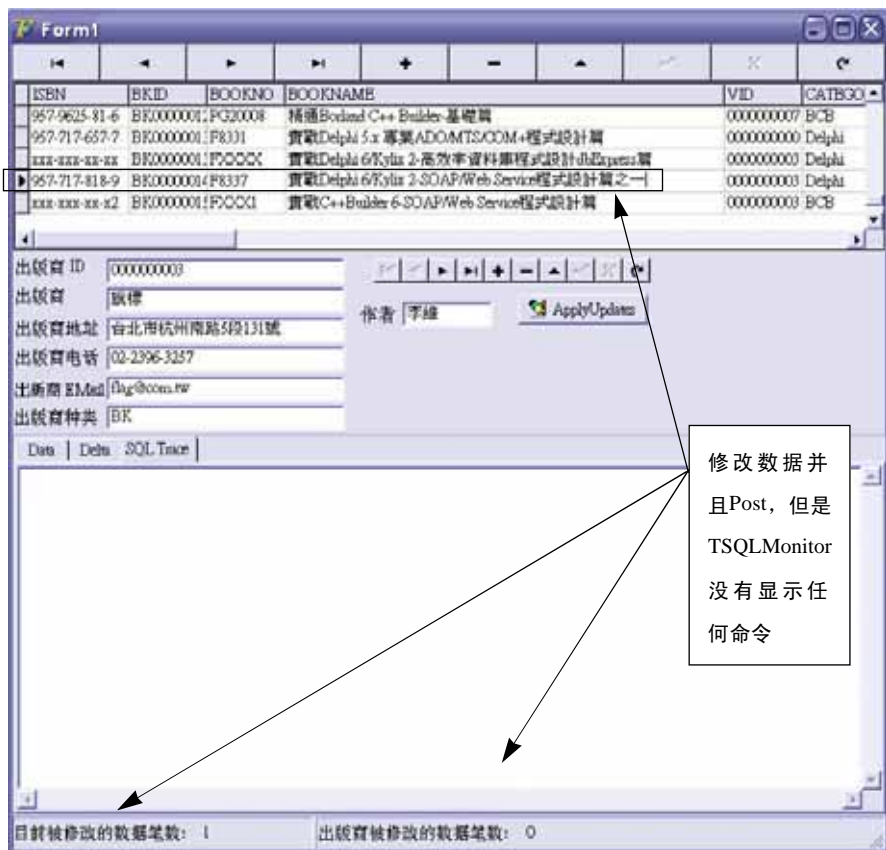


图2-48 修改范例应用程序的数据并且观察 dbExpress 的执行行为

2.6 结论

本章说明了如何使用 dbExpress 组件组中的 TSimpleDataSet、TSQLDataSet、TSQLQuery 以及 TSQLStoredProc 组件和 TSQLMonitor 组件。这些组件是程序员在开发数据库应用程序时经常需要使用的，因此切实掌握如何使用这些组件是非常重要的事情。

TSimpleDataSet 组件应该是程序员最常使用的组件了，因为它可以帮助程序员访问数据和处理数据的修改，此外它的许多特性值设置也会影响应用程序处理数据的行为和性能，因此 TSimpleDataSet 是程序员一定要切实掌握的 dbExpress 组件。事实上，TSimpleDataSet 组件只是融合了 TClientDataSet 和 TDataSetProvider 组件的功能，因此程序员也可以直接使用 TClientDataSet 和 TDataSetProvider 组件，只不过 TSimpleDataSet 组件简化了程序员需要进行的工作。TSimpleDataSet 组件处理数据的

行为依赖于DataSnap技术，因此本章也详细说明了 DataSnap处理数据的概念，让程序员能够精确地掌握DataSnap技术。



图2-49 应用范例应用程序中的数据更新并且观察 dbExpress的执行行为

除了TSimpleDataSet组件之外，许多数据处理工作也可以使用其他 dbExpress组件来完成，例如 TSQLDataSet或是 TSQLQuery组件。虽然这些组件无法处理修改数据的工作，但是却可以进行查询数据或是执行 DDL 语句的工作。

如果程序员需要执行后端数据源中的存储过程，那么 TSQLStoredProc组件便是非常好的工具。TSQLStoredProc组件能够轻易地取得数据源中所有存储过程的名称，并且让程序员选择要执行的存储过程，再通过Params特性值来传递存储过程需要的参数。

最后，本章说明了如何使用 TSQLMonitor组件来监视客户端的 dbExpress使用了什么SQL语句与后端数据源交互。通过使用 TSQLMonitor组件，程序员可以掌握客户端dbExpress的执行行为，也可以作为 SQL调试之用。在稍后的章节中也将会使用 TSQLMonitor组件来观察和调整 dbExpress应用程序的性能，因此 TSQLMonitor组件可以说是非常重要的dbExpress组件之一。

第3章 更多的dbExpress技巧

本书前面两章讨论的内容应该可以帮助读者顺利地使用 dbExpress来开发基本的数据库应用程序了。但是对于开发真正能够使用的数据库应用系统来说，程序员可能仍然需要许多额外的技巧，例如排序、搜寻数据、如何处理错误、数据库的事务管理等。

从本章开始将会一一介绍这些高级的 dbExpress技巧。读者在掌握了这些经常需要使用的dbExpress技巧之后，才能够轻松地了解后面章节介绍的深入概念和技术。本章将讨论的内容是属于比较容易上手的 dbExpress技巧，在下一章中将会详细讨论如何使用dbExpress来搜寻数据。读者在阅读完本章和下一章的内容之后，将能够切实掌握如何使用dbExpress处理数据。

3.1 数据排序

对数据进行排序是许多数据库应用程序都需要的功能，例如对顾客的身份证号排序、对订单金额排序或是对传票日期排序等。这些功能相信是许多数据库应用程序都经常需要使用的。除了简单的排序之外，许多应用程序也需要做进一步的排序工作。例如当应用程序对顾客的身份证号排序之后，又希望能够根据这个主排序再对顾客名称进行二级排序等。

当然，由于dbExpress使用SQL语句从后端数据源取得数据，因此刚才描述的功能都可以使用SQL语句的Order By和Group By等功能来实现，非常简单。但是对于dbExpress来说，要排序的数据可能已经在客户端的 TSimpleDataSet/TClientDataSet的Data特性值中了，如果使用SQL语句来重新排序的话，那么就等于浪费了已经存储在客户端的数据。因此对于许多排序的应用而言，可能只是需要对现有的数据排序，而不是从数据源中再次取得数据。如何使用 dbExpress实现这种应用呢？

本小节讨论的内容就在于说明如何使用 dbExpress进行排序的工作。通过讨论各种排序的方法，读者可以了解 dbExpress如何处理排序，并且根据读者自己的需求使用比较适合的方法。

在开始讨论之前，先让我们看看排列范例使用的数据模块。图 3-1是本小节范例应用程序使用的数据模块，这个数据模块使用了 TSimpleDataSet组件、TSQLDataSet、TDataSetProvider和TClientDataSet组件。这些组件在稍后的小节中都会一一讨论和使用。其中的scdsDemo和sdsDemo都使用select * from BIOLIFE从BIOLIFE数据表中

选取数据并且进行排序的工作。

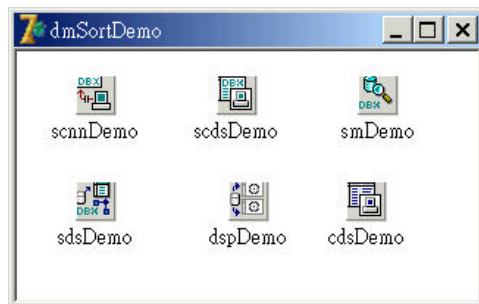


图3-1 本小节使用的范例数据模块

图3-2是范例应用程序使用的主窗体。这个范例应用程序将使用数种方法来进行排序的工作，并且会比较各种排序方法的异同和优缺点。



图3-2 范例应用程序的主窗体

现在我们就可以使用各种排序方式来讨论如何在应用程序中实现排序的功能。

3.1.1 dbExpress/DataSnap默认排序

当使用 TSimpleDataSet 组件选取数据时，TSimpleDataSet 便会自动地帮助程序员建立两个默认的排序方式，第一个是根据选择的数据表的主键建立的默认排序方式，第二个是允许程序员动态修改的排序方式。在 dbExpress 中分别以 DEFAULT_ORDER 和 CHANGEINDEX 来表示。例如，现在如果我们点击数据模块中 scdsDemo 的 IndexDefs 特性值的话，那么就可以看到如图 3-3 所示的画面，在 IndexDefs 特性值中

包含了DEFAULT_ORDER和CHANGEINDEX。如果我们继续点击 DEFAULT_ORDER，那么会在对象检视器中看到这个排序方式是以 SPECIES_NO字段为排序字段的。

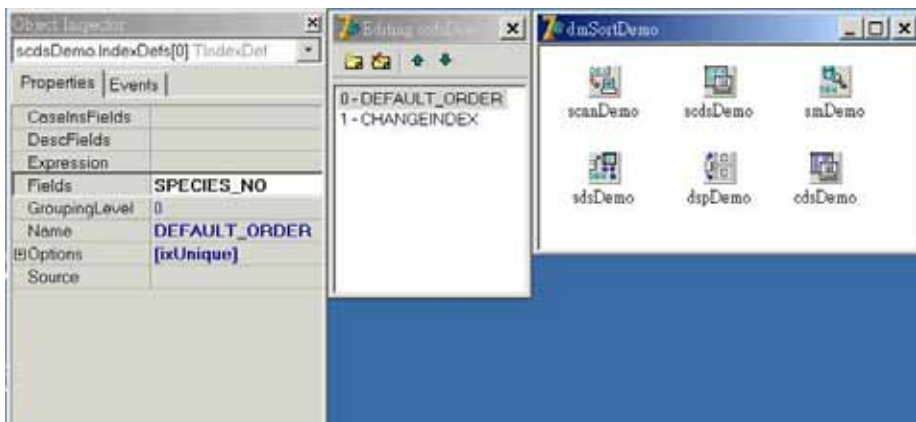


图3-3 TSimpleDataSet组件拥有默认的排序方式

那么为什么 DEFAULT_ORDER以SPECIES_NO字段为排序字段呢？如果我们开启BIOLIFE数据表的话，那么从图 3-4的画面中可以看到 SPECIES_NO字段是BIOLIFE数据表的主键字段，因此dbExpress会以这个字段做为默认的排序字段。

PK	Field Name	Field Type	Domain	Length	Sc...	SubT...	Array	Not N...	Charset	Co
1	SPECIES_NO	DOUBLE P.						<input checked="" type="checkbox"/>		
	CATEGORY	VARCHAR		10				<input type="checkbox"/>	BIG_5	BIC
	COMMON_NAME	VARCHAR		10				<input type="checkbox"/>	BIG_5	BIC
	SPECIES_NAME	VARCHAR		15				<input type="checkbox"/>	BIG_5	BIC
	LENGTH_CM	DOUBLE P.						<input type="checkbox"/>		
	LENGTH_IN	DOUBLE P.						<input type="checkbox"/>		
	TOPOTYPE	VARCHAR		10				<input type="checkbox"/>	BIG_5	BIC
	GRAPHIC	BLOB		80		Binary		<input type="checkbox"/>		

图3-4 范例数据表的纲要，SPECIES_NO是主键字段

因此，如果读者只需要使用主键字段做为排序字段，那么只需要在 TSimpleDataSet的IndexDefs特性值中选择DEFAULT_ORDER为排序方式即可。

但是对于没有定义主键的数据表（虽然这可能是有问题的设计），或是想要动态地使用其他字段来排序的话，那么应该如何做呢？

3.1.2 使用TSQLDataSet的排序特性

如果想要使用除了主键字段之外的其他字段来排序，那么可以使用 TSQLDataSet组件的SoftFieldNames特性值来指定。例如，现在我们希望范例应用程序使用非主键字段SPECIES_NAME做为排序字段，那么就可以使用对象检视器设置 TSQLDataSet

组件的SoftFieldNames特性值为SPECIES_NAME字段（见图3-5）。

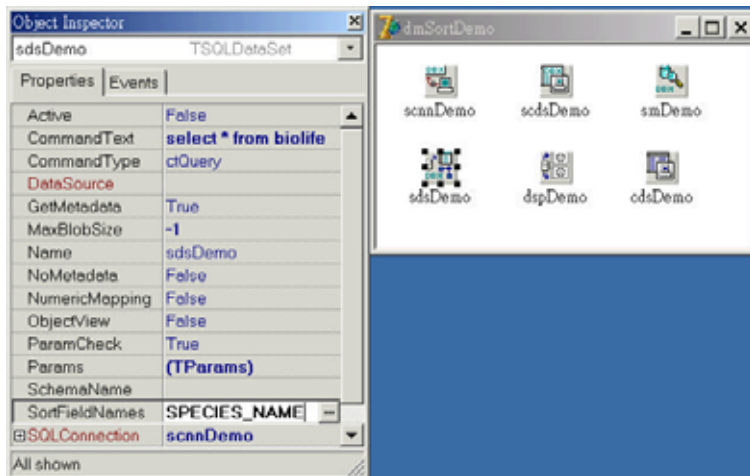


图3-5 TSQLDataSet的SoftFieldNames特性值允许程序员设置排序字段

基本上当程序员设置了 TSQLDataSet 的 SoftFieldNames 特性值后，当 TSQLDataSet 执行 CommandText 的 SQL 语句时会在 SQL 语句之后加上 Order By 子句，从下面由 TSQLMonitor 显示的结果可以看得出来：

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from biolife order by SPECIES_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
```

现在如果再次执行范例应用程序，那么我们就可以看到类似于图 3-6 的结果。从 TDBGrid 中可以看到现在数据表中的数据已经按照 SPECIES_NAME 字段的次序来排序了。

TSQLDataSet 的 SoftFieldNames 特性除了可以根据一个字段来排序之外，程序员也可以使用 SoftFieldNames 特性进行多个字段的排序。例如，现在除了希望以 SPECIES_NAME 字段作为主排序字段之外，我们还希望以 SPECIES_NO 字段做为第二级排序的依据。要进行这样的排序很容易，我们只要在 SoftFieldNames 特性中输入这两个字段，并且使用“,”符号分隔每一个字段名称即可。例如，在图 3-7 的对象检视器中的 SoftFieldNames 特性中输入了 SPECIES_NAME, SPECIES_NO 值，这代表同时使用这两个字段进行数据排序的工作。

现在再次执行应用程序并且使用 TSQLMonitor 进行观察，那么我们可以看到下面的 SQL 语句，这说明 dbExpress 在 order by 子句中使用了程序员在 SoftFieldNames 特性中输入的字段名称来进行数据排序。

```

INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from biolife order by SPECIES_NAME, SPECIES_NO
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind

```

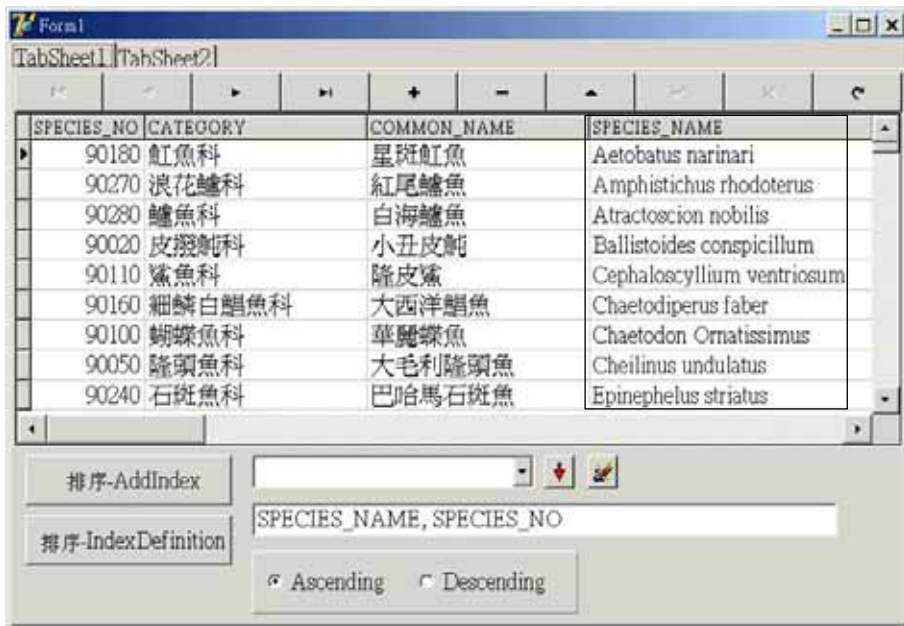


图3-6 在TSQLDataSet的SoftFieldNames特性被设置为排序字段之后，数据便会依据这个特性值来排序

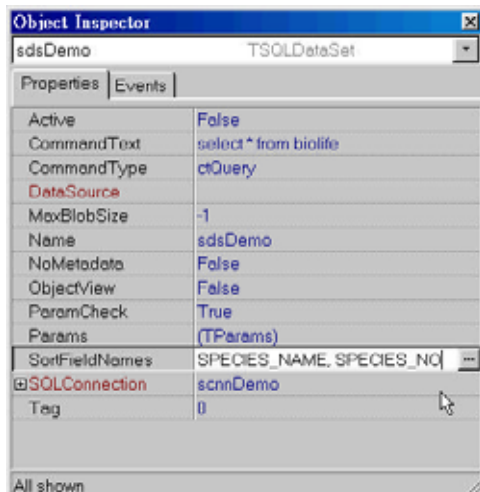


图3-7 在TSQLDataSet的SoftFieldNames特性中设置多个排序字段，每一个字段名称以“,”符号分隔

当然，除了使用 TSQLDataSet 的 SoftFieldNames 特性值之外，我们也可以直接使用 TSimpleDataSet 内建的 TInternalDataSet 的 SoftFieldNames 特性值来实现同样的效果，展开 TSimpleDataSet 的 DataSet 特性就可以看到 TInternalDataSet 的 SoftFieldNames 特性值。

3.1.3 在 TSimpleDataSet 中进行动态排序

前面讨论的排序技巧大都属于在应用程序设计时期指定排序条件，虽然很方便但是毕竟弹性不够大。在许多的数据应用中，应用程序需要对数据进行动态的排序。例如，在应用程序设计时期程序员在 SoftFieldNames 特性中使用了 SPECIES_NAME 字段做为排序条件，但是在应用程序执行时用户可能想使用其他字段来查看数据。这种在应用程序执行时期动态改变数据排序方式的需求是很常见的。那么 dbExpress 提供了什么方法帮助程序员在应用程序执行时期对数据进行动态排序呢？

dbExpress 基本上提供了两种方法让程序员可以在应用程序执行时期进行动态数据排序。第一种是使用 AddIndex，第二种是使用 IndexDefs。这两种不同的动态排序方式各有特点，在下面的小节中将会讨论如何使用这两种动态排序技巧。

1. 使用 AddIndex

TCustomClientDataSet 类的 AddIndex 方法可以为已经存在的数据表动态建立新的索引。在 AddIndex 方法中，程序员可以指定要作为索引的字段以及如何对这些索引字段的数据进行排序，下面是 AddIndex 的函数原型：

```
procedure AddIndex (const Name, Fields: string; Options: TIndexOptions; const DescFields: string = const CaseInsensitive: string = const GroupingLevel: Integer = 0);
```

AddIndex 接受数个参数，而这些参数和第 3 个参数 Options 有着相当密切的关系。第 3 个参数 Options 允许程序员指定使用什么规则来进行排序的工作，例如程序员可以指定数据是以升序、降序或是大小写不敏感的方式排列，或是字段中的所有数据都必须是唯一的值。

第 3 个参数 Options 可以指定如下的数值：

```
type
    TIndexOption = (ixPrimary, ixUnique, ixDescending, CaseInsensitive,
    ixExpression, ixNonMaintained
    TIndexOptions set of TIndexOption;
```

下面的表格总结了 TIndexOption 值代表的意义：

值	意 义
ixPrimary	此索引是此数据集的主索引对象
ixUnique	此索引是指字段中的数据都必须是唯一的，不能有重复的字段值

(续)

值	意 义
ixDescending	以降序方式排列数据
ixCaseInsensitive	以大小写不敏感的方式排列数据
ixExpression	这个索引是使用dBase表达式来定义的（此值只适合在dBase数据库中使用）
ixNonMaintained	这种索引在修改数据时不会同时自动修改索引对象的信息

如果程序员指定了第 2 个参数 Fields 的值，那么数据排序的方式就由第 3 个参数 Options 来决定。

如果第 3 个参数 Options 是 ixDescending，那么程序员在调用 AddIndex 时就必须指定第 4 个参数 DescFields 的值。DescFields 是所有需要作为索引的字段名称，每一个字段以分号（;）分隔。如果 Options 是 ixCaseInsensitive，那么 AddIndex 就必须指定第 5 个参数 CaseInsFields 的值，每一个字段也是使用分号（;）分隔。当 Options 是其他的值时，那么就必须指定第 2 个参数 Fields 的值。AddIndex 的第一个参数则是新索引的名称。

了解了如何使用 AddIndex 动态建立索引以便进行数据排序后，我们继续在前面的范例应用程序中加入使用 AddIndex 进行数据排序的能力。

首先双击图 3-6 中“排序-AddIndex”按钮，并且在它的 OnClick 事件处理函数中编写如下的程序代码：

```
procedure TForm1.btnAddIndexClick(Sender: TObject);
var
    sIndexName : String;
begin
    if (AlreadyIsIndex (INDEXID + cbFields.Text) then
        DeleteIndex (cbFields.Text);
    sIndexName := AddIndexForSCDS(cbFields.Text, rbAscending.Checked);
    dmSortDemo.scdsDemo.IndexName := sIndexName;
    Delete (sIndexName, 1, Length(INDEXID));
    slIndex.Add (INDEXID + sIndexName);
    dmSortDemo.scdsDemo.First;
end;
```

在上面的程序代码中，程序代码首先调用 AlreadyIsIndex 方法来检查用户在主窗体 edtSortFields 控件中输入的要建立索引的字段是否已经是索引字段了。如果是的话，就先调用 DeleteIndex 删除已经建立的索引对象，再准备重新根据用户输入的字段以及主窗体中选择的升序或是降序方式建立索引。

接着 btnAddIndexClick 函数调用 AddIndexForSCDS 方法，根据 edtSortFields 控件中的字段名称以及用户选择的排序方式来建立索引对象。在 AddIndexForSCDS 方法执行完毕之后，新的索引对象便已经建立了。最后，将数据模块中的 TSimpleData-

Set组件的IndexName设置为新建立的索引名称，以便使用新建立的索引来进行数据排序。

AlreadIsIndex方法会在slIndex中搜寻是否已经存在参数 sIndex指明的索引名称，并且返回搜寻的结果。

```
function TForm1.AlreadIsIndex (const sIndex: String): Boolean;
var
    iCount : Integer;
begin
    Result := False;

    for iCount := 0 to slIndex.Count do
    begin
        if (sIndex = slIndex.Strings[iCount]) then
        begin
            Result := True;
            Break;
        end;
    end;
end;
```

由于slIndex中存储的是索引的名称信息，用于避免用户在 TSimpleDataSet中建立重复的索引信息，因此在范例程序开始执行时我们先调用 TSimpleDataSet的GetIndexNames方法以便在slIndex中预先存储TSimpleDataSet的所有索引信息。同时调用FillFields在窗体中的TComBox控件中填入TSimpleDataSet的字段名称信息。

```
procedure TForm1.FormShow(Sender: TObject);
begin
    if (slIndex.Count = 0) then
        dmSortDemo.scdsDemo.GetIndexNames (slIndex);

    if (cbFields.Items.Count = 0) then
        FillFields (cbFields);
    edtSortFields.Text := dmSortDemo.sdsDemo.SortFieldNames;
end;
```

为TSimpleDataSet真正建立索引信息的方法则是 AddIndexForSCDS。在AlreadIsIndex方法成功地查明目前要建立的索引尚不存在之后，范例程序才调用 AddIndexForSCDS方法建立索引信息。AddIndexForSCDS方法会判断用户在窗体中选择的升/降序方式，并且调用AddIndex为TSimpleDataSet实际建立索引信息。

```
function TForm1.AddIndexForSCDS (const sIndex: String;
    bAscending: Boolean): Boolean;
begin
```

```

Result := INDEXID + sIndex;
if (bAscending) then
    dmSortdemo.scdsDemo.AddIndex (Result, sIndex, [])
else
    dmSortdemo.scdsDemo.AddIndex (Result, '', [ixDescending], sIndex
end;

```

最后，范例程序也提供了删除索引信息的功能。要删除 TSimpleDataSet中的索引信息，只需要调用 DeleteIndex方法即可。DeleteIndex方法接受的参数就是欲删除的索引名称。因此在范例程序的 DeleteIndex方法中就直接调用 TSimpleDataSet的 DeleteIndex方法。

```

procedure TForm1.DeleteIndex (const sIndex: String)
begin
    dmSortDemo.scdsDemo.DeleteIndex (INDEXID + sIndex);
end;

```

现在编译此范例程序并且执行它，让我们为数个字段动态建立索引并且观察执行的结果。图3-8和图3-9显示的就是在范例程序中为 SPECIES_NAME和LENGTH_CM字段建立索引的结果。请注意在这些画面中显示了我们不但可以正确地通过 AddIndex建立索引，同时AddIndex也可以正确地建立升序或是降序索引。

到目前为止，我们已经讨论了可以为 TSimpleDataSet/TClientDataSet进行数据排序的数种方法，接下来我们再讨论另外一种数据排序技巧。



图3-8 使用AddIndex动态地为SPECIES_NAME字段建立升序和降序索引



图3-8 (续)

2. 使用IndexDefs和IndexName

除了AddIndex之外，程序员还可以通过 TSimpleDataSet 的 IndexDefs 和 IndexName 特性值同时根据多个字段来进行数据的排序。使用 IndexDefs 和 IndexName 进行排序非常简单。首先程序员调用 IndexDefs 的 AddIndexDef 方法建立一个 TIndexDef 对象，

然后在 TIndexDef 对象中设置各种索引信息，例如索引名称、索引的升 / 降序方式、索引的目标字段等，和前面的 AddIndex 使用方式非常相像，最后再把 TIndexDef 对象的索引名称指定给 TSimpleDataSet 的 IndexName 特性值，那么 TSimpleDataSet 就会使用 TIndexDef 对象指定的索引方式来对其中的数据进行排序。

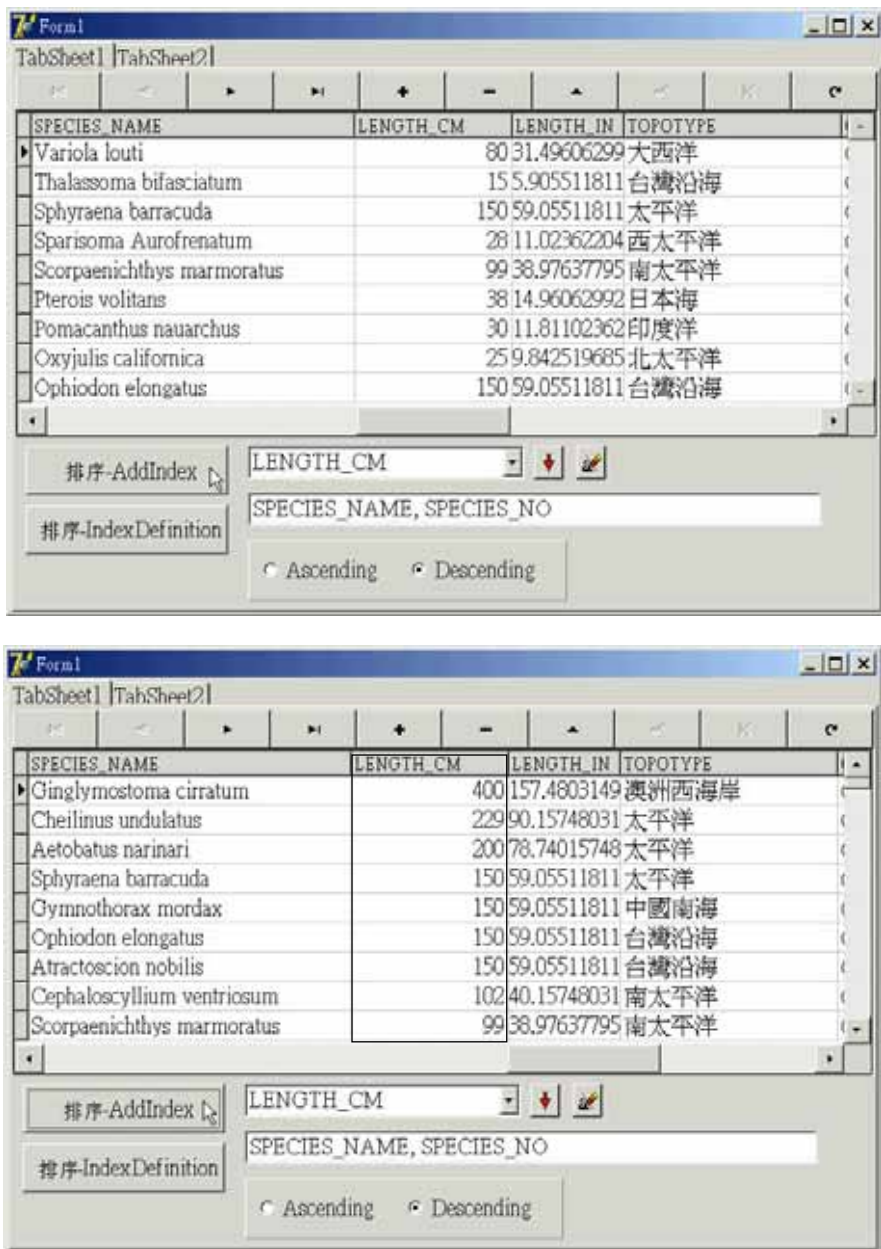


图3-9 使用AddIndex动态地为LENGTH_CM字段建立降序索引

图3-6中的“排序-IndexDefinition”按钮的事件处理函数就使用了 TSimpleDataSet/TClientDataSet的IndexDefs和IndexName特性值来进行动态数据排序的工作。在下面的“排序-IndexDefinition”按钮的事件处理函数中，先调用 AlreadIsIndexDef方法检查由窗体中 TEdit控件edtSortFields指定的字段是否已经建立了索引信息，如果没有的话就调用 CreateIndex方法来建立动态索引，最后把 CreateIndex建立的索引名称指定给数据模块中 TSimpleDataSet的IndexName值，以便对 TSimpleDataSet中的数据进行实际排序。

```
procedure TForm1.btnIndexDefsClick (Sender: TObject;
var
  sIndexName : String;
begin
  if (not AlreadIsIndexDef (edtSortFields.Text)) then
  begin
    sIndexName := CreateIndex(edtSortFields.Text, rbAscending.Checked
    dmSortDemo.scdsDemo.IndexName := sIndexName;
    dmSortDemo.scdsDemo.First;
  end;
end;
```

AlreadIsIndexDef方法使用IndexDefs的FindIndexForFields方法在 TSimpleDataSet中搜寻用户指定的字段是否已经建立索引了。如果用户已经使用相同的字段建立过索引，那么 FindIndexForFields会返回代表这些字段的 TIndexDef对象。在 FindIndexForFields方法检查之后，我们还需要调用 IndexDefs的Find方法进行相同的检查，只是这次检查是在字段名称之前加上 INDEXID这个代表索引前缀的字符串。

```
function TForm1.AlreadIsIndexDef (const sIndex: String: Boolean;
var
  adxf : TIndexDef;
begin
  Result := False;
  adxf := nil;

  try
    adxf := dmSortDemo.scdsDemo.IndexDefs.FindIndexForFields(sIndex);
    if (Assigned(adxf)) then
    begin
      Result := True;
      exit;
    end;
  except
    on exception do;
  end;
```

```

try
    adxf := dmSortDemo.scdsDemo.IndexDefs.AddIndexDef(INDEXID + sIndex;
    if (Assigned(adxf)) then
        Result := True;
    except
        on Exception do;
    end;
end;

```

如果 `AlreadyIsIndexDef` 没有搜寻到根据用户指定字段建立的索引信息，那么 `CreateIndex` 便会被执行以便实际建立索引。 `CreateIndex` 调用 `IndexDefs` 的 `AddIndexDef` 方法建立 `TIndexDef` 对象，再指定 `TIndexDef` 对象的 `Name` 特性值以设置索引名称，设置 `TIndexDef` 对象的 `Fields` 特性值以指定排序的字段名称，每一个字段名称也是使用分号（;）分隔的。 `TIndexDef` 对象的 `Options` 可以设置排序的特性，例如升序、降序或是大小写不敏感方式排序。最后 `CreateIndex` 返回建立的索引名称，以便指定给 `TSimpleDataSet` 的 `IndexName` 特性值。

```

function TForm1.CreateIndex (const sIndex: String; bAscending : Boolean; String;
var
    adxf : TIndexDef;
begin
    adxf := dmSortDemo.scdsDemo.IndexDefs.AddIndexDef;
    adxf.Name := INDEXID + sIndex;
    adxf.Fields := sIndex;
    if (not bAscending) then
        adxf.Options := [ixDescending];
    Result := adxf.Name;
end;

```

现在如果我们再次执行范例程序，那么就可以看到 `IndexDefs` 和 `IndexName` 对数据进行排序的效果了，如图 3-10 所示。

前面讨论了许多排序数据技巧，有的非常简单，只需要使用对象检视器设置即可，有的比较有弹性，但是需要使用程序代码来进行动态排序。不管使用哪一种方式似乎都可以达到数据排序的目的，非常方便。不过这是不是表示程序员可以自由选择自己喜欢的技巧来进行数据排序呢？数据排序有没有什么限制呢？

3.1.4 排序时考虑的因素

答案是“是的，程序员可以选择适合的技巧来进行数据排序，但是在排序时程序员必须注意数据排序实际的执行特性”。这是什么意思呢？简单地说， `TSimpleDataSet` 把 dbExpress 目前访问的数据存储在 `TSimpleDataSet` 管理的缓冲存储器中。由于



图3-10 使用IndexDefs进行多字段动态排序

TSimpleDataSet可以采用分段的方式访问数据，因此可能后端数据库中拥有大量的数据，而TSimpleDataSet目前只有少数的数据。但是在使用前面介绍的数据排序方法排列数据时，TSimpleDataSet会先把后端的所有数据读到客户端，再进行数据排序的工作。这会造成多种不利的后果：

- 把大量数据读到前端会造成网络的瞬间大负荷，造成所有客户端应用程序执行缓慢，降低网络的使用率。
- 客户端应用程序可能因为内存不足而造成应用程序错误。
- 需要花费大量的时间，造成应用程序反应迟钝。
- 这可能不是用户想要的结果。

用户可能只想对目前在 TSimpleDataSet 中的数据进行排序，而不是对后端的所有数据，因此这样做不但没有效率，而且用户也不会接受。现在让我们以一个实际的范例来说明。图3-11显示的就是我们描述的情形，在后端数据表中拥有 20000个记录，假设现在用户只想对目前在 TSimpleDataSet 中的数据进行排序，因此使用了下面的程序代码来进行数据排序。当然这些程序代码是使用前面介绍的排序技巧。

```
procedure TfrmPerfMain.btnAddIndexClick(Sender: TObject;
var
  sIndexName : String;
begin
  LogStartTime;
  // dmDBExpress.sqlmTest.Active := True;
  sIndexName := AddIndexForSCBDataSetSortFields.Text, rbAscending.Checked
  dmDBExpress.cdsTest.IndexName := sIndexName;
  ledtRecordCount.Text := IntToStr(dmDBExpress.cdsTest.RecordCount);
  LogEndTime;
  LogRunTime(mmSorts, 'AddIndex Sort' + IntToStr(UTL0OPS) + '笔数据时间 : ');
end;
```

但是在上面的程序代码执行之后，从图 3-11下方 TClientDataSet 显示的数据记录数中可以看到，这个范例程序已经把所有 20 000个记录读到了客户端，而且花费了 2.374秒，更麻烦的是用户如何在这 20 000个排序后的记录中找到他原先想要看到的数据？

因此程序员应该尽量避免直接对 TSimpleDataSet/TClientDataSet 进行数据排序的工作，以避免拖垮 dbExpress 客户端应用程序的性能。但是如果程序员真的只需要对目前存在于 TSimpleDataSet/TClientDataSet 中的数据进行排序的话，那么可以使用一点儿小技巧来达成目的，又不会从后端数据库中将所有数据读取到客户端。

这个技巧就是使用 TSimpleDataSet/TClientDataSet 的 ConeCursor 方法为 TSimpleDataSet/TClientDataSet 建立一个复制的游标管理机制，以切断 TSimpleDataSet/TClientDataSet 与后端数据源的连接，再进行排序。如此一来，当复制的游标进行数据排序时就不会从后端数据源中取得所有数据。 ConeCursor 方法有如下的原型：

```
procedure ConeCursor (Source: TCustomClientDataSet; Boolean;
KeepSettings: Boolean = False; virtual;
```

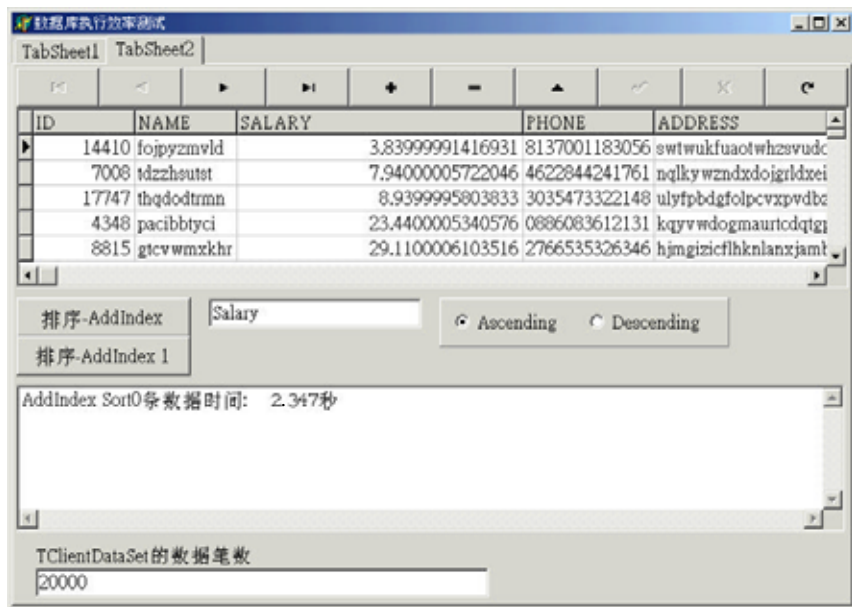



图3-11 TSimpleDataSet在进行数据排序时会把后端的所有数据读到前端，再进行排序工作，这会花费大量的时间

它的第一个参数 Source代表要复制的游标的源 TSimpleDataSet/TClientDataSet，第2个和第3个参数则代表复制的特性。Reset和KeepSettings参数可以控制下面列出的源 TSimpleDataSet/TClientDataSet组件以及目的地 TSimpleDataSet/TClientDataSet组件的特性值和事件处理函数：

- Filter、Filtered、FilterOptions和OnFilterRecord
- IndexName
- MasterSource和MasterFields
- ReadOnly
- RemoteServer和ProviderName

下面的表格描述了Reset和KeepSettings这两个参数的设置值对于上面列出的特性值和事件处理函数的影响：

Reset值	KeepSettings值	意 义
False	False	目的地TDataSet在前面列出的特性值都设置成与源TDataSet一样的值
True	N/A	目的地TDataSet在前面列出的特性值设置都会被清除
False	True	目的地TDataSet在前面列出的特性值都不会被改变

程序员可以通过设置Reset和KeepSettings这两个参数值来控制目的地 TSimpleDataSet/TClientDataSet的执行特性。

例如，下面的程序代码就是一个很好的范例。在下面的程序代码中，我们首先使

用一个暂时的 TSimpleDataSet/TClientDataSet组件 cdsTemp，并且调用它的 ConeCursor方法以复制原本拥有数据的 TSimpleDataSet/TClientDataSet组件 cdsTest。由于 cdsTemp只是复制 cdsTest的游标，因此与 cdsTest连接的后端数据源没有连接关系。在这之后，程序代码就只针对 cdsTemp管理的数据进行新增索引的工作来进行数据排序，如此一来就可以避免从后端数据源将大量数据读取到客户端的成本。

```
procedure TfrmPerfMain.Button1Click(Sender: TObject);
var
    sIndexName : String;
begin
    LogStartTime;
    dmDBExpress.cdsTemp.CloneCursor(dmDBExpress.cdsTest, True, False);

    sIndexName := INDEXID + edtSortFields.Text;
    if (rbAscending.Checked) then
        dmDBExpress.cdsTemp.AddIndex(sIndexName, edtSortFields.Text) []
    else
        dmDBExpress.cdsTemp.AddIndex(sIndexName, edtSortFields.Text, [ixDescending]);

    dmDBExpress.cdsTemp.IndexName := sIndexName;
    ledtRecordCount.Text := IntToStr(dmDBExpress.cdsTemp.RecordCount);
    LogEndTime;
    LogRunTime(mmmSorts, 'AddIndex Sort 1' + IntToStr(LGOPS) + '笔数据时间 : ');
end;
```

图3-12显示的是使用上面的技巧之后与图 3-11中相同的数据排序。从图 3-12可以看到使用 CloneCursor技巧之后，客户端的记录只有 100个，而且整个数据排序过程的时间减少到 0.01秒。当然这是因为现在我们只对目前存在于 TSimpleDataSet/TClientDataSet中的数据排序。

除了使用 CloneCursor之外，如果程序员希望排序的数据能够与原始数据之间有更清楚的分隔，那么程序员可以把拥有原始数据的 TSimpleDataSet/TClientDataSet组件的 Data特性值指定给另外一个独立的 TSimpleDataSet/TClientDataSet组件，再针对后一个 TSimpleDataSet/TClientDataSet组件进行数据排序。这是因为 TSimpleDataSet/TClientDataSet组件的 Data特性值包含的是目前存在于 TSimpleDataSet/TClientDataSet中的数据。

下面总结了程序员在使用 TSimpleDataSet/TClientDataSet进行数据排序时应该牢记在心的事情：

- 在默认情况下，当程序员针对 TSimpleDataSet/TClientDataSet中的数据进行处理时，TSimpleDataSet/TClientDataSet都会尝试把后端的所有数据读取到客户端。

端再进行处理，因此程序员必须注意这个执行行为带来的后果。

- 如果程序员只想对目前在 TSimpleDataSet/TClientDataSet 中的数据进行排序，那么就使用 CloneCursor 或是额外的 TSimpleDataSet/TClientDataSet 组件进行排序。
- 如果程序员真的想对所有数据进行排序，那么请直接改变 TSimpleDataSet/TClientDataSet 的 CommandText 特性值，使用 SQL 语句的 Order By 子句来取得排序的数据，让后端数据源来帮助进行数据排序。

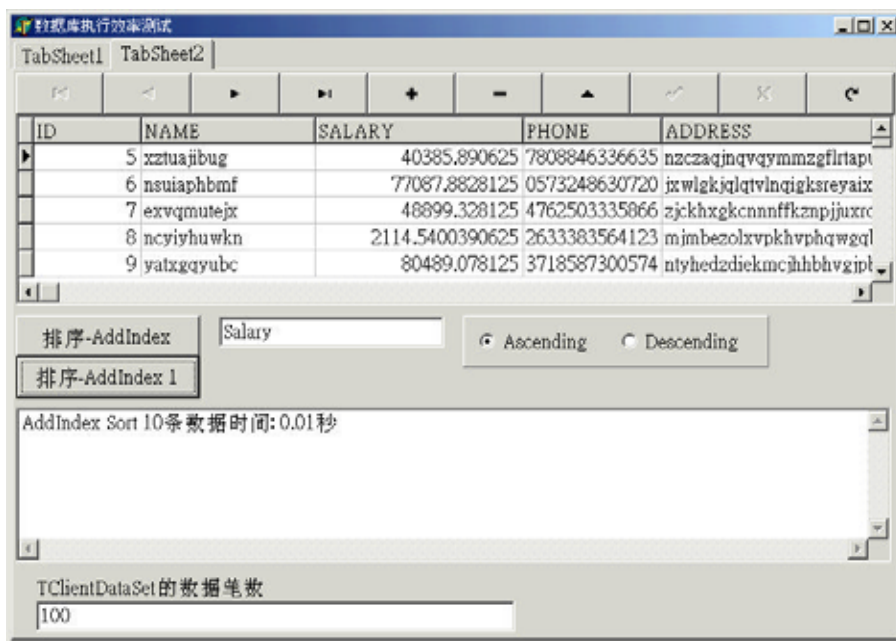


图3-12 通过小技巧可以避免 TSimpleDataSet 把所有数据读取到前端之后才排序数据

3.2 内存数据表

在开发应用系统时，经常会需要暂时存储一些应用程序执行时期的数据。这些数据可能不存在于数据库中，但是在应用程序执行时却需要经常引用这些数据。对于这种类型的数据以及应用而言，程序员可能会使用各种不同的方法来解决。不过 dbExpress 中的 TSimpleDataSet/TClientDataSet 组件却可以提供非常好的解决方案，如果程序员能够再结合上一小节讨论的排序技术，那么就可以在应用程序中实现快速的内存数据表（Memory Table），来进行暂时数据的存储以及对于这些暂时数据的频繁查询的工作。

例如，假设在一个范例应用程序中产生了下列表格中的暂时数据。这些暂时数据可能有数十个记录，而且在应用程序执行时经常需要根据 ID 来查询相应的字符串值。

ID (整数值)	值 (字符串)
1	DataSnap
2	WebSnap
3	Web Service
4	SOAP
5	XML
6	Interface
7	COM+
...	...

对于这种应用，程序员可以使用 TSimpleDataSet/TClientDataSet 建立内存数据表，然后根据 ID 字段排序，再让应用程序使用这个内存数据表来存储和查询数据。如此一来，不但可以解决存储暂时数据的问题，也可以很有效率地查询暂时数据。此外，使用 TSimpleDataSet/TClientDataSet 建立的内存数据表还可以存储多个字段值，也可以存储各种类型的数据，所有的内存数据表内容也可以动态产生，好处非常多。当然，TSimpleDataSet/TClientDataSet 的内存数据表也可以在应用程序执行完毕之后把这些暂时数据存储成 XML 格式的数据，等下次应用程序再次执行时直接从 XML 中加载这些暂时数据，而无需再次重新建立。本小节讨论的内容就是说明如何使用 TSimpleDataSet/TClientDataSet 来动态建立内存数据表，以便处理和查询暂时数据。

在下面的范例应用程序中将会使用 TSimpleDataSet 建立一个内存数据表，并且在这个内存数据表中填入应用程序经常需要使用和查询的数据。由于这些数据存储在内存中，因此查询的速度非常快，也不会增加数据库服务器的负荷。

步骤 1：设计范例应用程序

首先在 Delphi/Kylix 集成开发环境中建立一个新的应用程序项目，再建立一个数据模块。在数据模块中放入 TSQLConnection 以及两个 TSimpleDataSet 组件，如图 3-13 所示。

接着设置数据模块中的 dbExpress 组件特性值如下：

TSQLConnection:

特性名称	设置特性值
Name	scnnEmployee
Database	e:\Program Files\Common Files\Borland Shared\Data\Employee.gdb

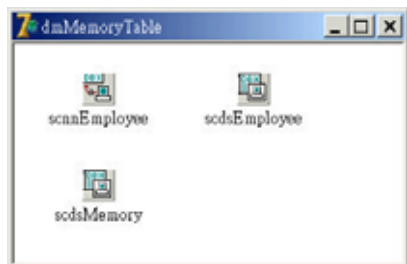


图3-13 范例应用程序的数据模块，使用两个 TSimpleDataSet 组件，其中 scdsMemory 是准备作为内存数据表使用的

TSimpleDataSet:

特性名称	设置特性值
Name	scdsEmployee
DBConnection	scnnEmployee
CommandText	select * from EMPLOYEE

TSimpleDataSet:

特性名称	设置特性值
Name	scdsMemory

请读者注意，在数据模块中的 scdsMemory中没有设置其他特性值，也没有连接到任何数据表。scdsMemory的内部结构以及数据会在稍后由范例应用程序动态地填入。此外在这个范例中是使用 InterBase 的范例数据库 Employee.gdb。

接着再设置范例应用程序的主窗体，如图 3-14所示。



图3-14 范例程序主窗体

在图3-14主窗体中有两个 TButton 组件，一个用来建立内存数据表以及填入应用程序使用的数据，另外一个 TButton 组件则让用户在 ID 控件中输入搜寻数据的键值之后搜寻内存数据表中的相应数据。主窗体下方的 TDBGrid 用来显示应用程序动态建

立的内存数据表，而主窗体右方的 TListBox则会显示建立内存数据表以及搜寻内存数据表花费的时间。

TButton:

特性名称	设置特性值
Name	btnCreateMemoryTable
Caption	产生暂时数据

TButton:

特性名称	设置特性值
Name	btnSearchData
Caption	搜寻暂时数据

设计完范例应用程序的控件之后，现在可以开始实现此范例应用程序，看看如何使用 TSimpleDataSet 建立内存数据表。

步骤 2: 实现范例应用程序

首先实现 btnCreateMemoryTable 按钮的 OnClick 事件处理函数。当用户点击了这个按钮之后，btnCreateMemoryTable 就会使用数据模块中的 scdsMemory 动态建立内存数据表。

下面即是 btnCreateMemoryTable 按钮的 OnClick 事件处理函数，首先它会开始计时以计算建立内存数据表和填入暂时数据需要花费的时间，接着它调用 CreateMemoryTable 函数以建立内存数据表的内部结构，例如字段名称、字段类型等元数据。在成功地建立了内存数据表之后，它会再调用 FillTempData 以便在内存数据表中填入应用程序需要使用的暂时数据。最后它显示完成这些工作花费的时间。

```
procedure TForm1.btnCreateMemoryTableClick(Sender: TObject);
begin
    StartTime;
    CreateMemoryTable;
    FillTempData;
    EndTime;
    ShowAppMsg('建立内存数据表和数据时期 : ' + FloatToStr(GetRunTime));
end;
```

CreateMemoryTable 是真正建立内存数据表的函数。在 CreateMemoryTable 中将会为数据模块中的 scdsMemory 建立字段信息以存储暂时数据，并且会建立索引字段以增加搜寻数据的速度。

CreateMemoryTable 首先调用 scdsMemory 的 FieldDefs 特性的 AddFieldDef 方法，以便在 scdsMemory 中建立暂时的字段定义对象， TFieldDef。当应用程序建立了字段

定义对象之后，必须设置这个字段定义对象的数据类型、字段长度以及字段名称等重要的特性值。从下面的程序代码中我们可以看到，CreateMemoryTable在scdsMemory中分别建立了两个字段定义对象，第一个字段定义对象是定义为整数类型的字段，这个字段的名称是TID。第二个字段定义对象是字符串类型的，它的字段长度是20个字符，而字段的名称是TSValue。

接着CreateMemoryTable又为scdsMemory定义了一个索引定义对象，TIndexDef。索引定义对象可以为数据集建立字段索引，程序员需要设置要建立索引的字段名称以及这个索引本身的名称。从下面的程序代码中可以看到，这个索引定义对象为TID字段建立索引，并且设置这个索引的名称是idxTID。

```
procedure TForm1.CreateMemoryTable;
begin
  with dmMemoryTable.scdsMemory do
  begin
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := 'TID';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 20;
      Name := 'TSValue';
    end;
    with IndexDefs.AddIndexDef do
    begin
      Fields := 'TID';
      Name := 'idxTID';
    end;
    CreateDataSet;
    IndexDefs.Update;
    IndexName := 'idxTID';
  end;
end;
```

当scdsMemory通过字段定义对象和索引定义对象设置要定义的字段以及索引之后，就可以调用 TSimpleDataSet的CreateDataSet方法建立一个不包含数据的新数据集了。最后记得调用 TSimpleDataSet的IndexDefs对象的Update方法以便反映最新的数据表索引信息。

FillTempData非常简单，它只是在scdsMemory建立的内存数据表中新增数个记录，其中第一个字段值是整数，第二个字段值是字符串数据。

```

procedure TForm1.FillTempData;
begin
    InsertData (1, 'DataSnap');
    InsertData (2, 'WebSnap');
    InsertData (3, 'Web Service');
    InsertData (4, 'SOAP');
    InsertData (5, 'XML');
    InsertData (6, 'Interface');
    InsertData (7, 'COM+');
end;

procedure TForm1.ApplyData;
begin
    dmMemoryTable.scdsMemory.ApplyUpdates (0);
end;

procedure TForm1.InsertData (const IID: Integer; const sValue: String);
begin
    with dmMemoryTable.scdsMemory do
    begin
        Insert;
        FieldByName ('TID').Value := IID;
        FieldByName ('TSValue').Value := sValue;
        Post;
    end;
end;

procedure TForm1.EndTime;
begin
    lEnd := GetTickCount;
end;

function TForm1.GetRunTime: double;
begin
    Result := (lEnd - lStart) / 1000.0;
end;

procedure TForm1.StartTime;
begin
    lStart := GetTickCount;
end;

procedure TForm1.ShowAppMsg (const sMsg: String);
begin
    lbTimes.Items.Add (sMsg);
end;

```

完成了 btnCreateMemoryTable 的 OnClick 事件处理函数之后， btnSearchData 的 OnClick 事件处理函数就简单多了。当用户点击了 btnSearchData 按钮之后，它就直接

使用scdsMemory刚才建立的内存数据表，调用Lookup方法来根据用户在主窗体中的ID控件中输入的ID键值搜寻数据。

```
procedure TForm1.btnSearchDataClick(Sender: TObject);
begin
    StartTime;
    edtValue.Text := tdmMemoryTable.scdsMemoryTable.IDObjectID.Text,
    'TSValue');
    EndTime;
    ShowAppMsg('搜寻数据时间 : ' + FloatToStr(GetRunTime));
end;
```

现在这个建立并使用内存数据表的范例应用程序已经完成了，让我们执行这个范例应用程序来观察执行的结果。

执行范例应用程序，点击“产生暂时数据”按钮建立内存数据表并且填入暂时数据，再在ID控件中输入ID值2搜寻数据，结果见图3-15。在主窗体右方的TListBox中可以看到，建立内存数据表并且填入暂时数据的速度非常理想。而搜寻内存数据表的数据更是快速无比，充分地显示了内存数据表吸引人的地方。



图3-15 使用内存数据表存储数据

内存数据表这种建立和搜寻速度快的特性非常适合在少量、经常使用的数据库应用中使用。不过读者必须注意，内存数据表不适合用来存储大量的暂时数据，例如超过数千个记录的情形就不适合使用。读者必须根据数据的特性以及客户端机器的内存情况适当决定是否要使用内存数据表。

3.3 使用计算字段

Delphi/Kylix为客户端的数据集提供了计算字段的功能，以帮助程序员建立在编写应用程序时需要的暂时字段。所谓计算字段（Calculated Field）是指在许多数据库应用程序中经常需要使用一些暂时的字段来存储一些数值，以方便应用程序进行计算工作。由于这些字段只是在应用程序执行时帮助应用程序完成工作，而不永久存储在数据库中。对于这类的应用，计算字段便是很好的解决方案。

由于DataSnap原本就是在客户端的内存中建立数据集结构，因此DataSnap当然也可以在这个内存结构中建立额外的不存在于数据表中的暂时字段。Delphi/Kylix提供了方便的UI和事件处理函数帮助程序员使用计算字段来解决数据库应用程序的需求。在UI方面，程序员可以通过激活数据集组件的字段编辑器来加入任意数量的计算字段，而且能够建立任何类型的暂时字段。另外，数据集的AutoCalcFields特性值可以控制计算字段的执行行为，OnCalcFields事件处理函数则是程序员使用计算字段的地方。在OnCalcFields事件处理函数中，程序员可以使用Object Pascal程序代码来为计算字段进行任何计算。OnCalcFields事件处理函数的原型如下：

```
type TDataSetNotifyEvent = procedure (DataSet: TDataSet of object;
```

其中参数DataSet是指触发此事件的数据集，也就是拥有此计算字段的数据集。

由于计算字段的弹性非常大，因为程序员可以定义任意数量和类型的计算字段，又可以使用Object Pascal在OnCalcFields事件处理函数中进行任何计算工作，因此计算字段在许多数据库应用程序中经常使用。而在Delphi/Kylix中使用计算字段是非常简单的，只需要完成下面的两个操作即可：

- 使用数据集的字段编辑器加入计算字段或是其他类型的暂时字段。
- 在OnCalcFields事件处理函数中使用程序代码执行计算字段的工作。

现在就让我们使用一个简单的范例来说明如何使用简易的计算字段帮助应用程序进行暂时的运算工作。在这个范例中，我们希望能够了解为员工加薪之后每一个员工的薪水，以决定加薪的幅度。由于这个应用只是满足决策者暂时的需求，而不需要把每一个加薪计算的暂时结果存储在数据表中，因此计算字段在这个应用中是非常适合的。因此我们将在数据集中建立一个暂时的浮点类型的计算字段以存储每一次的计算结果。

首先是激活目标数据集组件的字段编辑器以准备新增计算字段（图 3-16）。

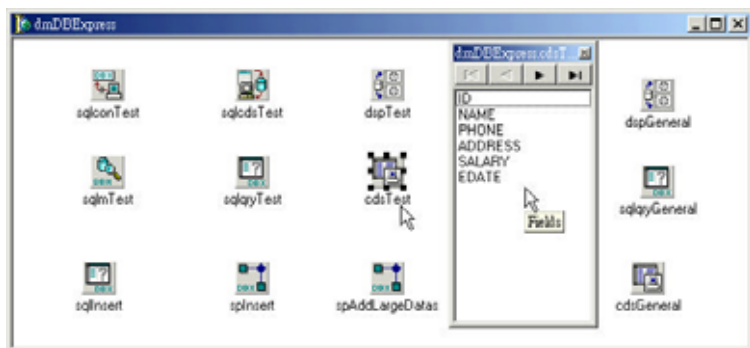


图3-16 激活TClientDataSet的字段编辑器

接着点击鼠标右键激活快捷菜单，从其中选择 New field...项目以激活 New Field 对话框，定义要建立的字段对象种类，见图 3-17。



图3-17 在字段编辑器中激活快捷菜单，选择 New Field...以建立暂时的字段

此时，Delphi/Kylix会显示如图 3-18所示的对话框让程序员定义各种不同类型的暂时字段对象，定义字段的种类以及字段名称等特性。例如，在图 3-18的对话框中便定义了一个类型为浮点数，名称为 AdjustedSalary的计算字段。

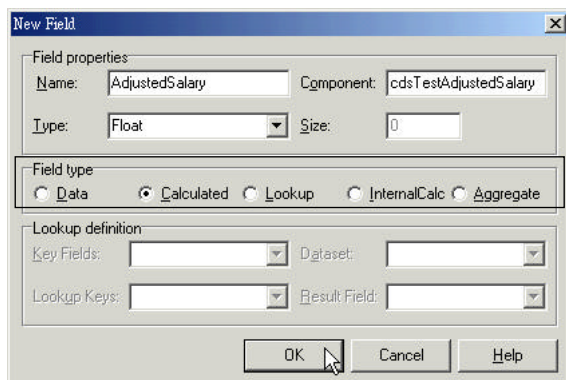


图3-18 在New Field对话框中建立计算字段，设置其名称和字段类型

每一个数据集允许建立的暂时字段并不相同，下面是 Delphi/Kylix允许建立的暂时字段类型，但是有一些暂时字段只能在特定的数据集中使用。例如 TSimpleDataSet便没有提供InternalCalc和Aggregate类型的暂时字段，但是 TClientDataSet提供了所有类型的暂时字段。

字段类型	目 的
Data	取代现有数据表中的字段
Calculated	计算字段，在 OnCalcFields事件处理函数中动态计算字段值
Lookup	用来查询信息的查询字段
InternalCalc	客户端数据集使用的内部计算字段，用来暂时存储数据的字段。与 Calculated字段不同，InternalCalc并不在OnCalcFields事件处理函数中计算
Aggregate	Aggregate类型的字段，下一小节将会说明

在图3-18中定义了AdjustedSalary计算字段之后，我们便可以在包含这个计算字段的数据集组件的 OnCalcFields事件处理函数中编写如下的程序代码在应用程序执行时动态地计算此计算字段的结果，也就是根据不同的加薪幅度来计算每一个员工的薪资信息：

```

procedure TdmDBExpress.cdsTestCalcFields (DataSet: TDataSet;
begin
    try
        DataSet.FieldByName ('AdjustedSalary').Value :=
            DataSet.FieldByName ('Salary').Value *
                (1 + StrToFloat(frmPerfMain.leDtAdjustedPercent.Text) / 100.0);
    except
        on Exception do;
    end;
end;

procedure TfrmPerfMain.bbbtnAdjustClick (Sender: TObject;
begin
    dmDBExpress.cdsTest.Edit;
    dmDBExpress.cdsTest.Cancel;
end;

```

上面的程序代码先取出真正存储员工薪资的字段的价值，再乘以用户在应用程序中输入的加薪百分比，并且把计算的结果存储在计算字段 AdjustedSalary中。请读者注意，当在数据集中加入了计算字段之后，程序员便可以按照使用数据表中的真正字段的方式来访问和使用计算字段。

图3-19便是范例程序使用计算字段的画面。从图3-19中读者可以发现计算字段对于数据感知组件来说就像是真正的数据表字段一样，可以显示或是进行任何修改。程序员当然也可以再根据用户对计算字段的修改使用程序代码来决定如何进行额外的处理。

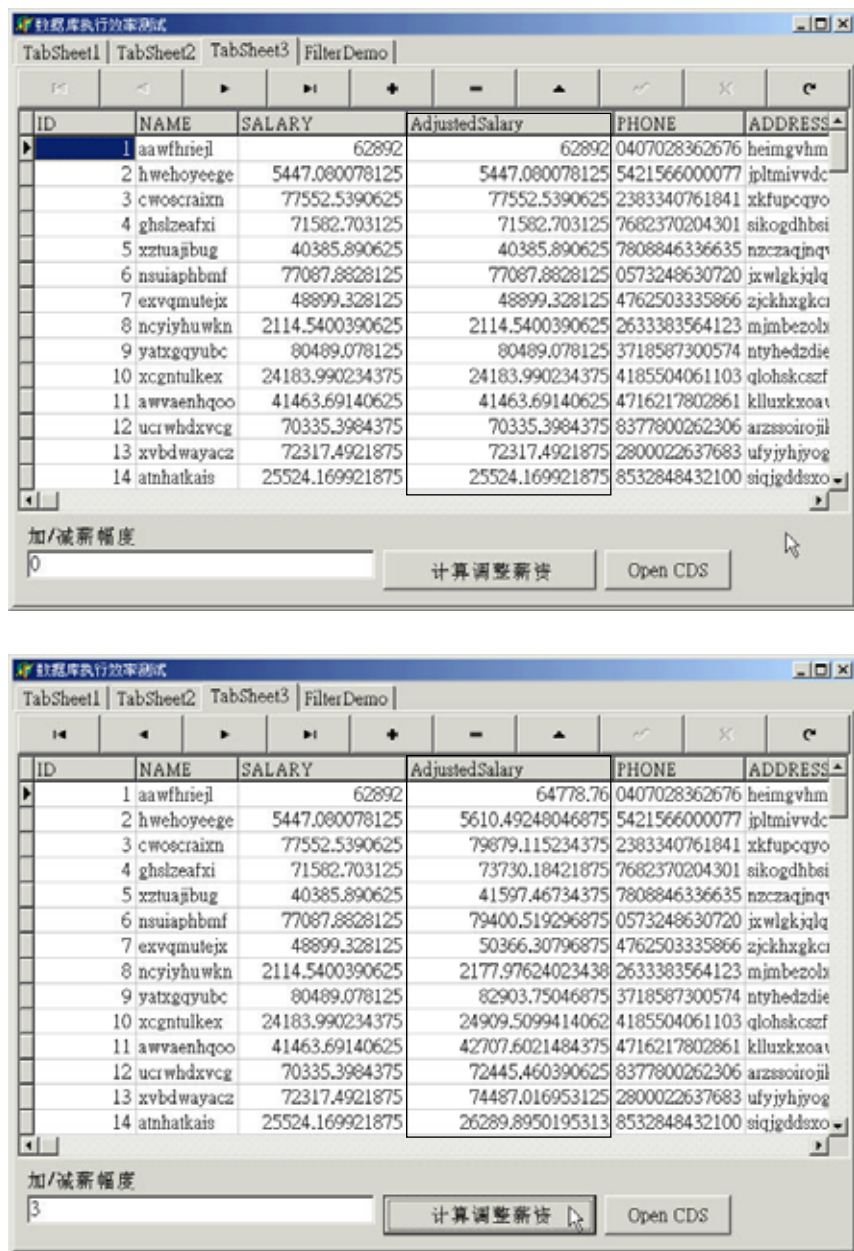


图3-19 计算字段执行的状态

虽然计算字段在使用上很方便，又很具弹性，但是数据集在许多情况下都会触发 OnCalcFields 事件处理函数，这造成了应用程序大量的计算负荷，因此读者必须注意不要在 OnCalcFields 事件处理函数中执行太复杂的计算，以避免应用程序的性能被计算字段拖垮。

3.4 使用Aggregate字段

Delphi/Kylix定义了一种特别的字段称为Aggregate字段，从Aggregate的名称便可以了解到它的目的是用来计算客户端数据集中数据的各种特殊值。程序员可以使用Aggregate计算客户端数据集中数据的总和、平均值或最大和最小值。Aggregate字段对象允许程序员使用表达式，并且使用Aggregate预先定义的一些操作数来帮助程序员进行计算的工作，这些预先定义的操作数总结在下面的表格中：

操 作 数	使用意义
Sum	计算数值字段的总和
Avg	计算数值字段或是日期字段的平均值
Count	计算非空字段的总数量
Min	取得数值、日期或是字符串字段的最小值
Max	取得数值、日期或是字符串字段的最大值

代表Aggregate字段的类是TAggregate类，TAggregate类中有一个Expression特性值，在Expression中程序员可以使用上面表格定义的操作数来进行计算的工作。使用Aggregate字段是非常简单的，而且我们在前面图 3-18中已经看过了Aggregate字段。因此要使用Aggregate字段，程序员只需采用下面的两个步骤即可：

- 使用字段编辑器定义Aggregate字段。
- 在Aggregate字段的Expression特性值中使用操作数输入要执行的表达式。

之后，程序员就可以在程序代码中访问Aggregate字段，就像访问一般的字段一样。一旦程序员访问Aggregate字段，Aggregate字段就会执行它的Expression特性值中定义的表达式，执行的结果就会成为Aggregate字段的值。

现在让我们继续使用上一小节中的范例来说明如何使用Aggregate字段，现在我们除了为每一位员工计算加薪之后的薪资之外，还希望得到Salary字段的总和。这可以使用Aggregate字段轻易地做到，因为Aggregate字段提供的sum操作数可以计算Salary字段的总和。

因此我们先点击客户端数据集组件，接着在对象检视器中点击Aggregates特性值激活Aggregate编辑器，然后加入一个新的Aggregate字段，如图3-20所示。

接着在Aggregate字段的Expression特性值中输入如下的表达式：

```
Sum (Salary)
```

这个表达式使用Aggregate字段提供的sum操作数来计算Salary字段值的总和。接着我们只需要在程序代码中直接访问这个Aggregate字段，就像访问一般的字段那样，这样就可以取得Aggregate字段代表的Salary字段值的总和：

```
procedure TfrmPerfMain.bbbtnAdjustClick(Sender: TObject);  
begin
```

```
dmDBExpress.cdsTest.Edit;  
dmDBExpress.cdsTest.Cancel;  
  
Self.ledtTotalAdjustedSalary.Text :=  
    dmDBExpress.cdsTest.Aggregates.Items[0].Value;  
end;
```

最后我们再次执行上一小节的范例，那么当我们输入加薪的百分比之后，下方的TEdit控件就显示了计算出的Aggregate字段，代表所有薪资的总和（见图3-21）。

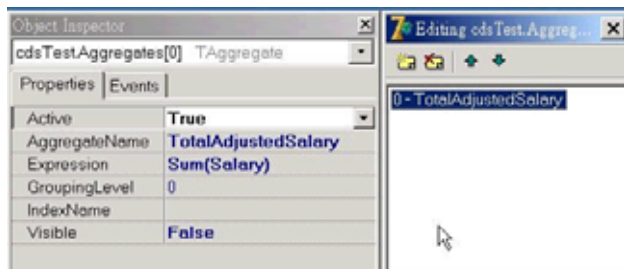


图3-20 建立Aggregate字段

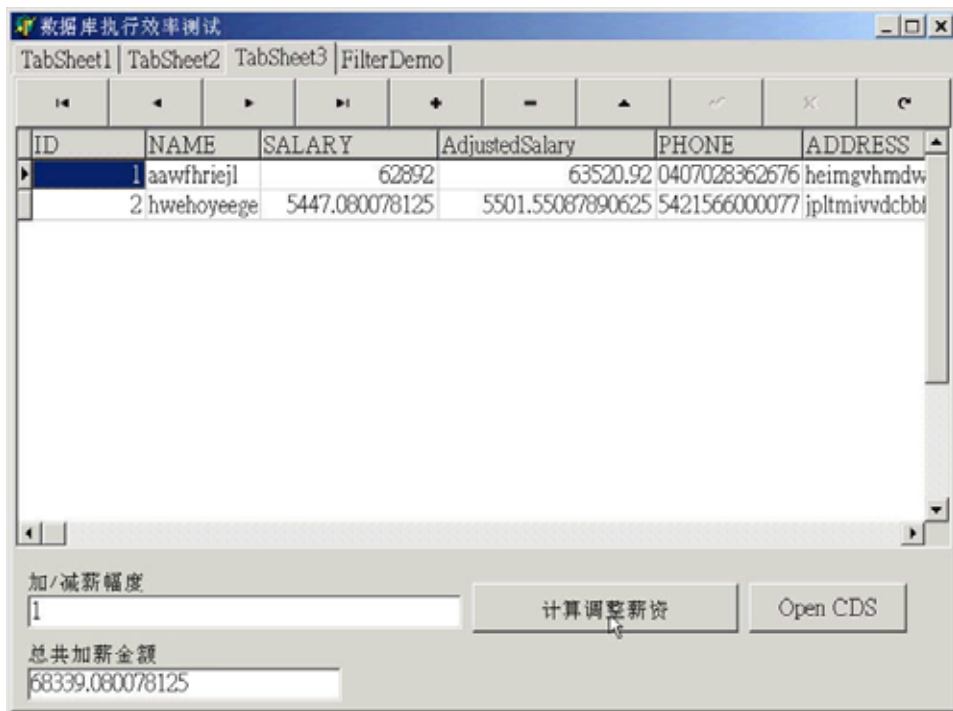


图3-21 Aggregate字段执行的状态

Aggregate字段是Delphi/Kylix提供的简易表达式功能，并且允许程序员把计算出的信息以字段的形式暂时存储在数据集中。不过程序员如果需要比较复杂的计算，

那么仍然应该使用 Object Pascal 程序代码或是计算字段。

3.5 UpdateStatus

TSimpleDataSet/TClientDataSet 的 UpdateStatus 特性值让程序员能够得知目前的数据的状态。例如，这个记录是未经过修改的原始数据，或是已被修改的数据，或是已被用户删除的数据，亦或是由用户新增的数据。下面的表格列出了 UpdateStatus 能够拥有的特性值以及每一个特性值代表的意义：

UpdateStatus 数值	意 义
usUnmodified	目前这个记录尚未修改过
usModified	目前这个记录已经被修改过
usInserted	目前这个记录属于新增的数据
usDeleted	目前这个记录已经被删除了

程序员可以直接使用程序代码检查 TSimpleDataSet/TClientDataSet 的 UpdateStatus 特性值来判断目前记录的状态，以便根据需要来处理数据。例如下面的程序代码可以把目前 TSimpleDataSet/TClientDataSet 中每一个记录的状态显示在主窗体中：

```

procedure TfrmUpdateStatus.NotifyScroll;
var
    aUS : TUpdateStatus;
begin
    aUS := dmUpdateStatus.sdsUpdateStatus.UpdateStatus;

    case aUS of
        usUnmodified : SetUpdateStatusInfo(False, False, False, False);
        usModified   : SetUpdateStatusInfo(True,  True, False, False);
        usInserted   : SetUpdateStatusInfo(False, False, True,  False);
        usDeleted    : SetUpdateStatusInfo(False, False, False, True);
    end;
end;

procedure TfrmUpdateStatus.SetUpdateStatusInfo (ck1, ck2, ck3,
    ck4: Boolean);
begin
    Self.cbUnModified.Checked := ck1;
    Self.cbModified.Checked := ck2;
    Self.cbInserted.Checked := ck3;
    Self.cbDeleted.Checked := ck4;
end;

procedure TfrmUpdateStatus.BitBtn1Click (Sender: TObject;

```

```

begin
  if (dmUpdateStatus.sdsUpdateStatus.ChangeCount) then
  begin
    dmUpdateStatus.sdsUpdateStatus.ApplyUpdates (0);
    NotifyScroll;
  end;
end;

```

如果执行上面的程序代码，在用户浏览每一个记录时，这个记录的状态便会显示在下方相对应的复选框中，见图 3-22。



图3-22 范例程序可通过UpdateStatus了解每一个记录的状态

请读者注意，UpdateStatus特性值显示的信息是目前存在于客户端 TSimpleDataSet/TClientDataSet缓存内存中的数据的状态。一旦 TSimpleDataSet/TClientDataSet调用了ApplyUpdates方法把数据更新回数据源中，TSimpleDataSet/TClientDataSet便会调用MergeChangeLog把客户端的Data和Delta合并，并且清除UpdateStatus特性值。

```
procedure MergeChangeLog;
```

因此，此时 TSimpleDataSet/TClientDataSet中所有数据的UpdateStatus特性值会被重置成usUnmodified。另外读者要注意的是，UpdateStatus特性值只会记录一个记录相对于原始情况的状态改变，而不会连续追踪记录的状态改变。这个意义是说，当在 TSimpleDataSet/TClientDataSet中新增一个记录时，其 UpdateStatus特性值被设置成usInserted；如果稍后这个记录又经过数据修改，那么 UpdateStatus特性值将仍然保持usInserted，而不会同时记录这个记录经过了 usInserted和usModified两个状态的改变。

3.6 SavePoint

在DataSnap提供的功能中有一个非常好的功能便是，当用户使用 TSimpleDataSet/TClientDataSet/TSQLClientDataSet在客户端修改数据时，DataSnap会记录用户对于数据的每一次修改。由于DataSnap会记录数据修改的过程，因此DataSnap也允许用户把数据恢复到先前的状态。提供这个功能的就是 TSimpleDataSet/TClientDataSet/TSQLClientDataSet的SavePoint特性值。

SavePoint和数据源提供的事务管理是不一样的，SavePoint特性值虽然可以允许程序员把数据的修改恢复成先前的状态，但这是对在客户端 TSimpleDataSet/TClientDataSet/TSQLClientDataSet缓存内存中的记录而言，并不是指已经通过ApplyUpdates更新回数据源的数据。

使用SavePoint特性值虽然可以恢复先前对数据的修改，但是程序员只能以向前的方式恢复数据，一旦使用SavePoint把数据恢复成上一次的状态，那么就无法再以向后的方式恢复数据。另外，当程序代码调用了ApplyUpdates方法之后，所有先前的SavePoint便失效了而进入重置的阶段，在随后的数据修改中SavePoint才又开始起作用。

使用SavePoint特性值是非常容易的，程序员只要在应用程序需要的状态下存储 TSimpleDataSet/TClientDataSet/TSQLClientDataSet的SavePoint特性值，接着在应用程序随后的阶段想要恢复数据状态时，再把先前存储的SavePoint值指定回SavePoint特性值即可恢复先前的数据。

例如，下面的程序代码在 TSimpleDataSet的AfterPost事件处理函数中把 TSimpleDataSet的SavePoint特性值存储在一个 TListBox中。而在 bbtnSavePointClick事件处理函数中则从 TListBox中取出用户选择的先前存储的SavePoint值，再指定给 TSimpleDataSet的SavePoint特性值，以把数据恢复成当时阶段的状态。

```
procedure TForm1.sdsBooksAfterPost (DataSet: TDataSet;
begin
    ListBox1.Items.Add (IntToStr (Self.sdsBooks.SavePoint));
end;

procedure TForm1. bbtnSavePointClick(Sender: TObject;
var
    iSavePoint : Integer;
begin
    if (ListBox1.ItemIndex <>)-1 then
    begin
        iSavePoint := StrToInt(ListBox1.items[ListBox1.ItemIndex]);
        Self.sdsBooks.SavePoint := iSavePoint;
    end;
end;
```

```
procedure TForm1. bbtnApplyClickSender: TObject;  
begin  
    if (Self.sdsBooks.ChangeCount > 0) then  
        Self.sdsBooks.ApplyUpdates (0) ;  
end;
```

图3-23～图3-25是执行上面程序代码的结果。首先我们修改 FXXX、FXXX1和 FXXX3这3个记录的BOOKNAME字段的值，在每一次Post数据之后，SavePoint的值会被保存在TListBox中。因此我们在图3-23中看到了三个数值：65537、65538和65539。

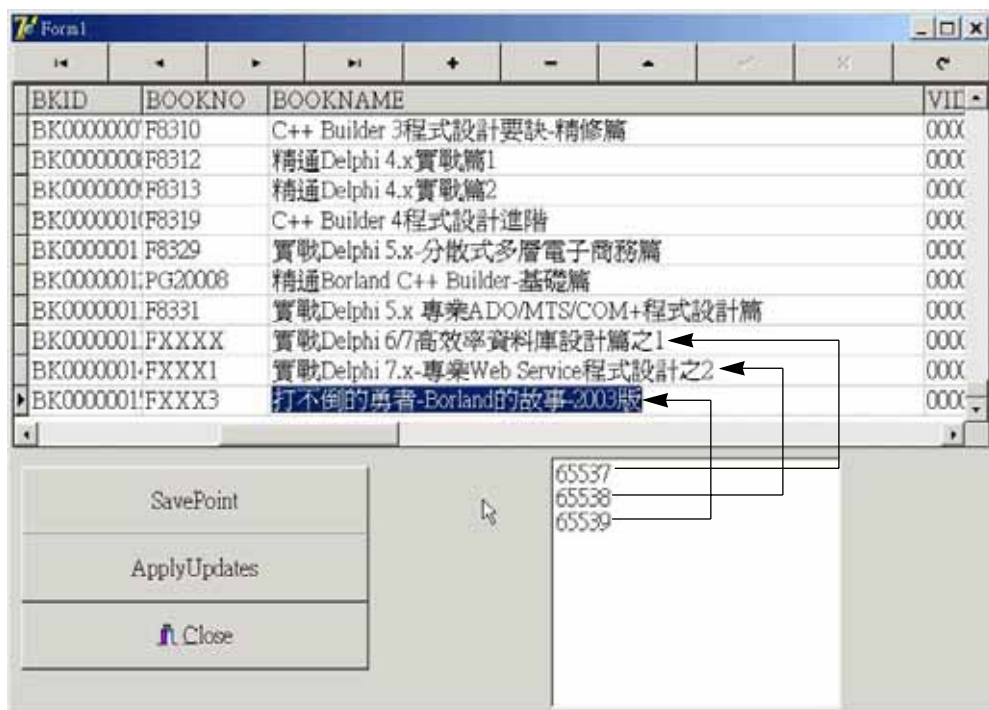


图3-23 修改3个记录并且存储了三个 SavePoint 值

如果我们点击 TListBox 中的 65538，再点击窗体中的“SavePoint”按钮，那么我们可以看见类似于图 3-24 的画面，此时 TSimpleDataSet 的数据已经恢复到存储第 3 个 SavePoint 之前的状态。我们对于 FXXX3 这个记录的修改已经被恢复成未修改之前的状态了。

如果我们接着又点击 TListBox 中的 65539，再点击“SavePoint”按钮想要向后恢复数据的状态，那么 DataSnap 便会显示一个 Invalid Parameter 错误对话框，这说明了 SavePoint 是只能向前恢复状态，而不能向后恢复数据状态。同样，在调用了 ApplyUpdates 方法把客户端的所有修改数据更新回数据源之后，所有先前的 SavePoint 值也无效了，如果程序代码试着使用先前的 SavePoint 值，那么也会遇到 Invalid Parameter

错误对话框（见图3-25）。

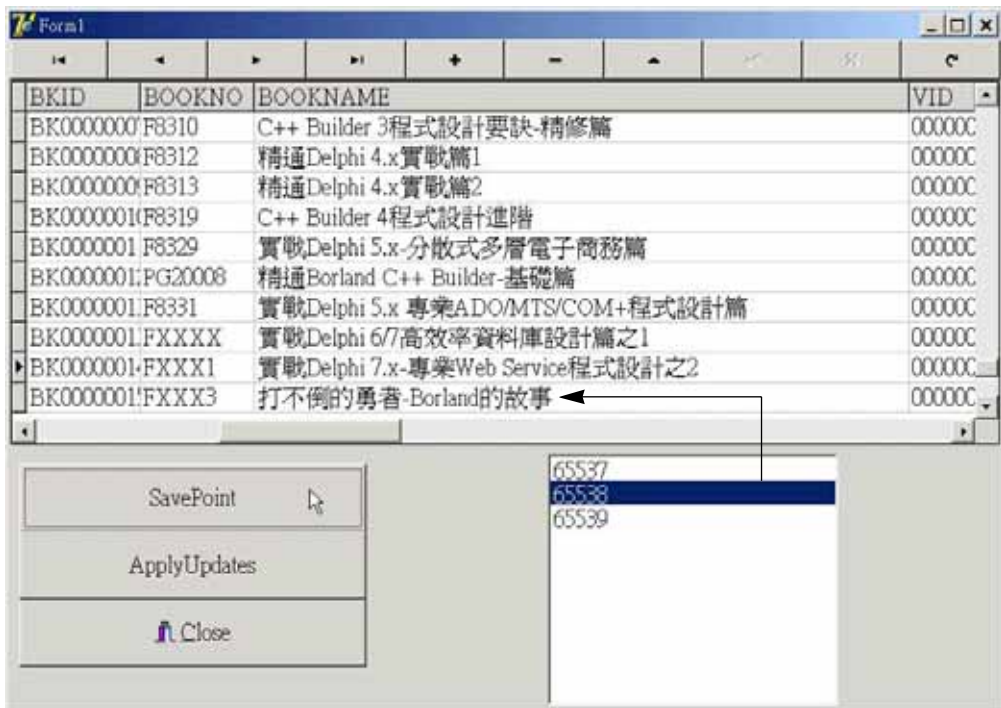


图3-24 点击第二个SavePoint值以恢复当时的数据状态



图3-25 使用向后方式恢复 SavePoint或是在 ApplyUpdates之后使用 SavePoint都会造成 Invalid Parameter错误

3.7 MyBase

Delphi的MIDAS/DataSnap技术从推出之后便支持 BriefCase模式的数据处理能力。所谓BriefCase模式是指客户端应用程序在使用DataSnap技术从数据源取得了数据之后，可以暂时把数据存储于客户端的机器中，切断和数据源的连接，再继续处理数据。当客户端处理完毕之后，又可以连接数据源，再把客户端修改的数据更新回数据源中。

这种数据处理模式非常适合用于把数据带出企业，在客户端处理数据，最后再回到企业中更新在客户端处理的数据。例如，许多销售人员需要在见顾客之前先从系

统下载产品信息，到顾客处展示，修改数据，最后再回到公司把最新的数据更新回企业的系统中。

在Delphi 6/7和Kylix中，Borland又再次扩充BriefCase数据处理模式，允许程序员使用XML的格式在客户端存储数据，或是与其他系统以XML的格式交换数据，因此把这种功能称为MyBase。

在TSimpleDataSet/TClientDataSet中提供了两个方法允许程序员把数据暂时存储在文件中，以及再从文件中加载先前存储的数据，它们是SaveToFile和LoadFromFile。SaveToFile除了可以存储数据之外，也允许程序员使用不同的格式来存储数据，程序员可以将数据存储为二进制格式或是XML格式。下面是SaveToFile的原型声明：

```
SaveToFile (const FileName: string = ''; Format: TDataPacketFormat=dfBinary
```

SaveToFile的第一个参数是存储的文件名称，第二个参数则是程序员指定的存储格式，在默认情况下是存储为二进制格式。下面的表格列出了 TDataPacketFormat能够拥有的值以及这些值代表的意义：

TDataPacketFormat	意 义
dfBinary	以二进制格式存储TSimpleDataSet/TClientDataSet中的数据
dfXML	以XML格式存储TSimpleDataSet/TClientDataSet中的数据
dfXMLUTF8	以XML, UTF8格式存储TSimpleDataSet/TClientDataSet中的数据

LoadFromFile方法更简单了，它只接受一个代表要加载的数据文件名的参数：

```
procedure LoadFromFile (const FileName: string = '');
```

如果程序员使用XML格式存储数据的话，那么就可以与其他应用程序交换数据。程序员可以使用Delphi 6/7提供的XML Data Binding向导来进行数据交换的工作。下面的程序代码片断显示了如何使用 SaveToFile和LoadFromFile方法来存储和加载TSimpleDataSet/TClientDataSet中的数据。

```
procedure TForm1.BitBtn3Click (Sender: TObject;
begin
    if (Self.rbtnBinary.Checked) then
        Self.sdsPerformers.SaveToFile (TFILENAME)
    else
        begin
            Self.sdsPerformers.SaveToFile (TFILENAME+ '.XML', dfXMLUTF8
        end;
end;

procedure TForm1.BitBtn4Click (Sender: TObject;
begin
    Self.sdsPerformers.Active := False;
    if (Self.rbtnBinary.Checked) then
        Self.sdsPerformers.LoadFromFile (TFILENAME)
    else
```

```
Self.sdsPerformers.LoadFromFile (TFILENAME + '.XML');  
end;
```

图3-26显示了当用户在 TSimpleDataSet 中修改了数据之后通过调用 SaveToFile 把数据暂时存储在一个 XML 文件中。

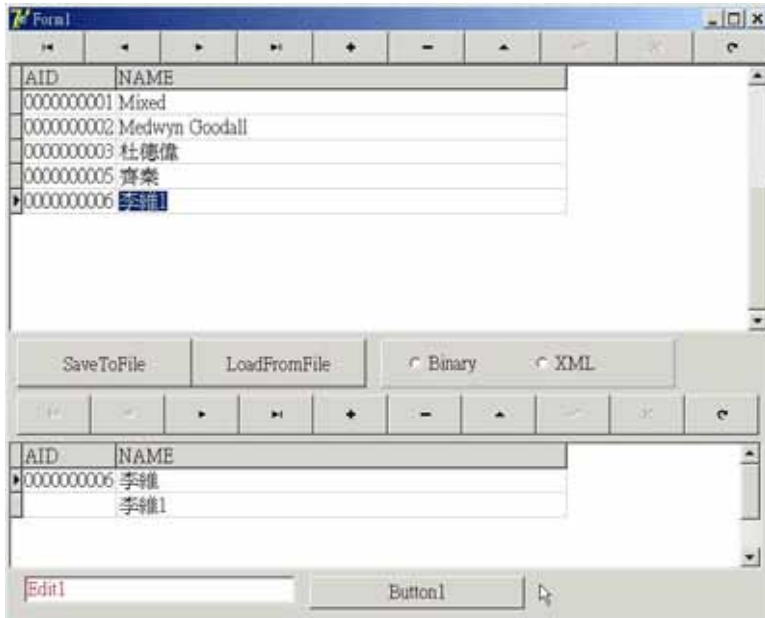


图3-26 将 TSimpleDataSet 中的数据存储到 XML 文件中

图3-27则是使用 IE 显示图3-26存储为 XML 格式的数据的画面，从图3-27中我们可

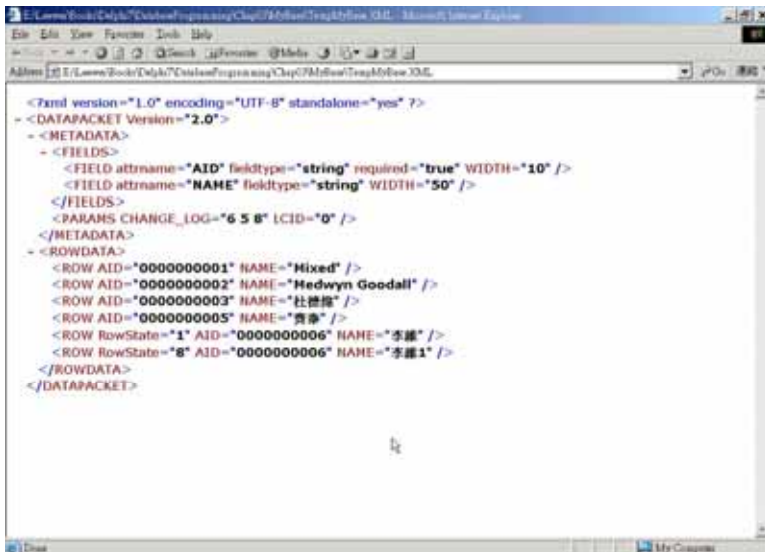


图3-27 IE显示DataSnap存储的XML格式的数据

以看到DataSnap的确是把所有数据正确地存储为XML格式，如此一来这个文件中的数据就可以顺利地与其他应用程序交换了。

虽然Delphi6/7的DataSnap使程序员能够将数据存储到文件中，以后再将数据加载回来，但是目前Delphi 7中的DataSnap在这方面似乎有一些小bug，在将数据加载回来之后没有正确地处理Delta特性值，在这稍后Delphi 7的Patch中也许会被纠正。

3.8 TField对象的SetText和GetText事件处理函数

在许多数据库设计中，分析师都会使用键值来代表特定的信息，例如使用ID值来代表部门代号，使用书籍号码来代表书名等。这些技巧有几个目的，其中最重要的目的通常是在数据表中维持唯一性，以便于搜寻特定的数据。其他目的则可能是为了使数据库正规化（不是所有的分析师都了解数据库正规化），也有可能是为了节省数据库存储空间等考虑因素。

虽然上述的设计是非常正确的，但是用户在使用应用程序时却可能不希望看到对于用户来说是无意义的键值代号，而是希望看到完整的信息。例如用户可能希望看到完整的书籍名称，而不是书籍代号，除非这个用户是书商，能够熟记每一个书籍代号代表的书籍，因此解决这个数据库信息转换问题的工作是许多数据库应用程序都需要处理的工作。

要解决这个问题有许多不同的方法，Delphi/Kylix的TField对象也提供了两个相关的事件处理函数SetText以及GetText来帮助程序员解决这个问题。GetText事件被触发的时机是当客户端需要取得字段的值时，因此对于键值的字段程序员可以在这个事件处理函数中使用键值到其他数据表中搜寻这个键值代表的完整信息，再把此完整信息作为字段的值返回即可。下面是GetText的声明原型：

```
type TFieldGetTextEvent = procedure (Sender: TField; var Text: String;  
    DisplayText: Boolean) of object;
```

TFieldGetTextEvent的第二个参数Sender是目前要取得值的字段对象，第二个参数Text是声明为var类型的字符串参数，程序员必须在这个参数中指定代表此字段的值，最后一个参数DisplayText则是代表这个字段是用来显示的字段，还是允许用户修改的字段。

如果程序员使用GetText事件处理函数转换字段代表的值，那么通常程序员需要搭配使用SetText事件处理函数。因为当GetText改变字段代表的值之后，如果用户在客户端修改了这个字段的数据，那么在用户把数据更新回数据源中时，程序员必须先把数据转换回原始的形式，然后再更新到数据源中。而进行这个数据回转的好地方就是SetText事件处理函数。

SetText事件的声明原型如下:

```
type TFieldSetTextEvent=procedure (Sender:TField; const Text:String of
object;
```

其中的第一个参数 Sender代表要更新数据的字段, 而 Text则是客户端传递来的代表此字段的值, 程序员可以将参数 Text的值恢复成正确的数据, 再把这个正确的数据指定给 Sender的 Value特性值。

现在让我们使用一个范例来说明如何使用 GetText和SetText事件处理函数。在 D7Books数据库中, BOOKS数据表有一个字段 AID, 它代表书籍的作者或是唱片的歌手ID。当我们使用数据感知组件显示 BOOKS数据表时, 我们并不希望看到 AID的值, 而是希望看到 AID代表的人名。因此这个范例在使用 TDBGrid显示BOOKS数据表的内容时, 使用 GetText和SetText事件处理函数将AID转换成人名, 并且在用户修改数据时将人名回转成正确的AID值, 再更新回BOOKS数据表中。

首先建立一个 Delphi/Kylix项目, 在数据模块中放入如图 3-28所示的dbExpress组件以连接D7Books数据库。

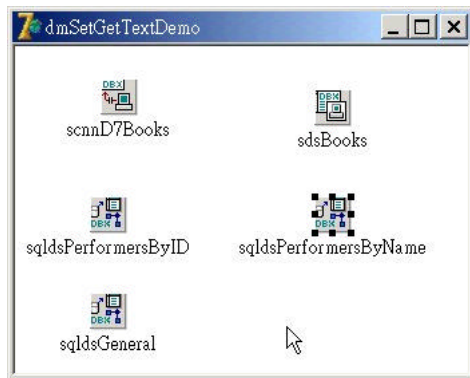


图3-28 使用dbExpress连接D7Books

TSQLConnection:

特性名称	设置特性值
Name	scnnD7Books
Database	e:\LeeWei\Books\Delphi7\Datas\D7Books.GDB

TSimpleDataSet:

特性名称	设置特性值
Name	sdsBooks
DBConnection	scnnD7Books
DataSet\CommandText	select * from BOOKS

TSQLDataSet:

特性名称	设置特性值
Name	sqlsPerformersByID
DBConnection	scnnD7Books
CommandText	select NAME from PERFORMERS where AID = :AID

TSQLDataSet:

特性名称	设置特性值
Name	sqlsPerformersByName
DBConnection	scnnD7Books
CommandText	select AID from PERFORMERS where NAME = :NAME

TSQLDataSet:

特性名称	设置特性值
Name	sqlsGeneral
DBConnection	scnnD7Books

接着激活 sdsBooks 的字段编辑器加入所有字段，并且使用对象检视器为 AID 字段定义 GetText 和 SetText，如图 3-29 所示。

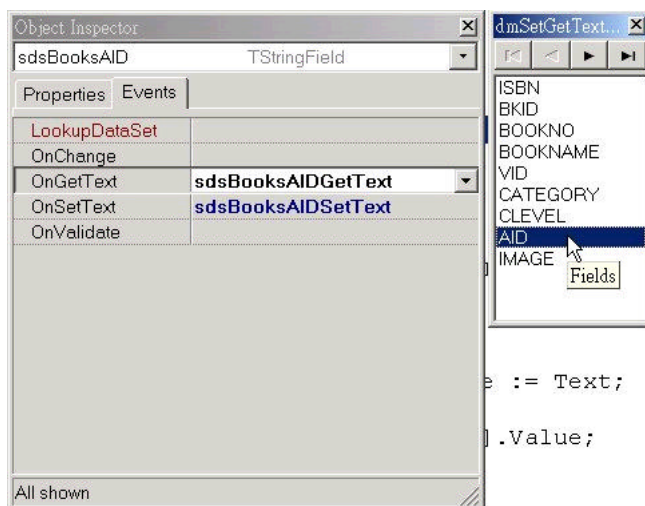


图3-29 激活 sdsBooks 的字段编辑器，加入所有字段，再建立 AID 字段的 GetText 和 SetText 事件处理函数

现在我们希望使用 TDBGrid 显示 BOOKS 数据表的内容，并且希望 AID 字段能够显示一个下拉框，允许用户在修改此字段的数据时能够使用选择人名的方式。因此，在主窗体中加入一个 TDBGrid 控件，双击 TDBGrid 激活字段编辑器，加入所有的字

段，如图3-30所示。由于我们希望在此字段中能够让用户选择人名，因此我们只需要把人名的数据加入到 AID 的 PickList 特性值中即可。

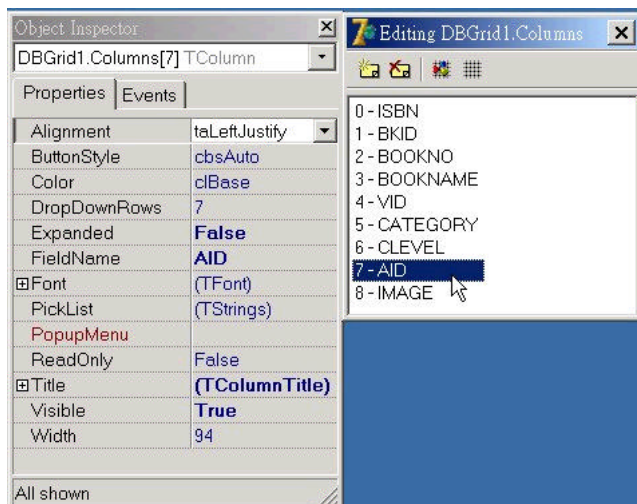


图3-30 为AID字段加入PickList特性值

首先我们实现数据模块中 AID 字段的 GetText 和 SetText 事件处理函数。下面是相关的实现程序代码：

```
procedure TdmSetGetTextDemo.sdsBooksAIDGetText (Sender: TField;
var Text: String; DisplayText: Boolean
begin
    try
        Self.sqldsPerformersByID.ParamByName ('AID') .Value := Sender.Value;
        Self.sqldsPerformersByID.Active := True;
        Text := Self.sqldsPerformersByID.Fields[0].AsString;
    finally
        Self.sqldsPerformersByID.Active := False;
    end;
end;

procedure TdmSetGetTextDemo.DataModuleCreate (Sender: TObject;
begin
    Self.sqldsPerformersByID.Prepared := True;
    Self.sqldsPerformersByName.Prepared := True;
end;

procedure TdmSetGetTextDemo.DataModuleDestroy (Sender: TObject;
begin
    Self.sqldsPerformersByID.Active := False;
```

```

Self.sqlldsPerformersByID.Prepared := False;
Self.sqlldsPerformersByName.Active := False;
Self.sqlldsPerformersByName.Prepared := False;

Self.scnnD7Books.Connected := False;
end;

procedure TdmSetGetTextDemo.sdsBooksAIDSetText (Sender: TField;
const Text: String;
begin
try
Self.sqlldsPerformersByName.ParamByName ('NAME').Value := Text;
Self.sqlldsPerformersByName.Active := True;
Sender.Value := Self.sqlldsPerformersByName.Fields[0].Value;
finally
Self.sqlldsPerformersByName.Active := False;
end;
end;

```

在sdsBooksAIDGetText事件处理函数中，我们根据 AID字段目前的数值到 PERFORMERS数据表中搜寻代表此 ID的人名信息，再把找到的人名指定给第二个参数Text。而sdsBooksAIDSetText事件处理函数则是执行相反的工作，它根据传递来的人名数据到PERFORMERS数据表中搜寻此人名的 AID值，最后再指定给代表字段对象的Sender参数的Value特性值。

由于GetText和SetText事件处理函数执行得非常频繁，因此为了提高性能，我们在数据模块的 OnCreate 事件处理函数中先准备sqlldsPerformersByID和sqlldsPerformersByName组件要执行的SQL语句，并且在数据模块的 OnDestroy 事件处理函数中取消准备，以释放数据源中使用的资源。

最后再回到范例程序的主窗体，在主窗体中主要的工作就是在显示 TDBGrid时在 AID字段的 PickList中填入目前在 PERFORMERS数据表中的所有人名信息。这个工作是在范例主窗体的 OnShow事件处理函数中调用 FillPerformerInfos函数完成的。而 FillPerformerInfos函数则是使用数据模块中的 sqlldsGeneral组件执行SQL语句从 PERFORMERS数据表中取得所有人名数据，再填入到 TDBGrid中AID字段的 PickList特性值中。

```

procedure TForm1.BitBtn2Click (Sender: TObject;
begin
if (dmSetGetTextDemo.sdsBooks.ChangeCount > 0) then
dmSetGetTextDemo.sdsBooks.ApplyUpdates (0) ;
end;

procedure TForm1.FillPerformerInfos;

```

```

begin
    if (Self.DBGrid1.Columns.Items[7].PickList.Count > 0) then
    begin
        dmSetGetTextDemo.sqldsGeneral.Active := False;
        dmSetGetTextDemo.sqldsGeneral.CommandText := 'select distinct NAME from
        PERFORMERS';
        try
            dmSetGetTextDemo.sqldsGeneral.Active := True;
            while not dmSetGetTextDemo.sqldsGeneral.EOF
            begin
                Self.DBGrid1.Columns.Items[7].PickList.Add (dmSetGetTextDemo.
                sqldsGeneral.Fields[0].AsString);
                dmSetGetTextDemo.sqldsGeneral.Next;
            end;
        finally
            dmSetGetTextDemo.sqldsGeneral.Active := False;
        end;
    end;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    FillPerformerInfos;
end;

```

最后我们编译此范例并且执行它，那么用户便可以看到类似于图 3-31 的画面，请注意此时 AID 字段显示的不再是代表人名的 ID 数值，而是完整的人名数据，而且用户可以使用下拉框来进行数据修改的工作。如果用户修改了 AID 字段中的人名数据并且点击主窗体中的 ApplyUpdates 按钮，那么数据模块中的 SetText 事件处理函数便会被触发，它先把人名回转成正确的 ID 值，再更新回 BOOKS 数据表的 AID 字段中。



图3-31 范例程序执行的画面

当然本范例只是为了展示如何使用 GetText和SetText这两个事件处理函数。对于这个范例来说，PERFORMERS数据表中的数据并不多。对于这种需要查询的数据表而言，如果数据量不多的话，那么程序员可以使用一个 TSimpleDataSet/TClientDataSet把数据先读到客户端，再在 GetText和SetText事件处理函数中直接使用 TSimpleDataSet/TClientDataSet的Locate方法进行数据转换的工作，这可能会比较有效率。

3.9 结论

在本章中我们讨论了许多使用 dbExpress的技巧，它们包含各种数据排序技巧以及如何高效率地进行数据排序，读者在熟悉了不同的排序方法之后，就可以根据应用程序的需求选择适当的排序方式。

内存数据表是一项非常有用的技巧，特别是对于应用程序频繁使用的数据。由于内存数据表把数据暂时存储在内存中，因此应用程序在处理或是查询这些数据时会非常高效。dbExpress的结构也特别适合使用内存数据表，因为 dbExpress本来就是一个离线的数据集，数据原本就存储在客户端的内存中，此外 dbExpress也允许应用程序动态地建立数据集结构并且建立索引，可以让内存数据表更有效率。不过读者必须知道，内存数据表虽然好用，但是内存数据表并不适合处理大量的数据，例如上千个记录就不适合。如果程序员有需要频繁处理的数据并且在数百个记录左右，那么就可以考虑使用内存数据表。

在本章最后讨论了其他许多有用的技巧，包括计算字段、SavePoint以及TField对象的事件处理函数等。这些处理数据的技巧在许多场合都非常有帮助，读者在了解了这些技巧之后，可以使用它们更方便地开发数据库应用程序。

第二部分

dbExpress进阶功能篇



第4章 搜寻数据

在前面的章节中，本书讨论了如何使用 dbExpress 组件从数据源中取得应用程序需要的数据。通过 dbExpress 的 TSQLDataSet 或是 TSimpleDataSet 组件，程序员可以从数据库中取得任何数据。当从数据源将数据取到客户端之后，这些数据便存储在客户端的内存中并且形成一个结果数据集（Result Dataset）。当程序员需要在这个结果数据集中搜寻数据时，就必须使用 TClientDataSet 或是 TSimpleDataSet 组件的搜寻方法来找到特定的数据。

TClientDataSet/TSimpleDataSet 组件提供了数个不同的方法让程序员能够在结果数据集中搜寻特定的数据，这些方法包括 Locate、Lookup、过滤器、SetRange 等。每一种搜寻数据的方法都有其特定的用途，程序员可以选择最适合的方法来搜寻数据。

当程序员使用 Locate、Lookup 等方法搜寻数据时，TClientDataSet/TSimpleDataSet 组件提供了强大的功能让程序员搜寻特定的数据，程序员可以根据单一字段值来搜寻数据，也可以同时根据多个字段来搜寻数据。此外，被搜寻的字段也不必是索引字段，即使是非索引字段也一样可以使用这些方法搜寻数据。

当 TClientDataSet/TSimpleDataSet 组件使用 Locate、Lookup 方法搜寻数据时，这些方法会自动使用最有效率的方法来搜寻数据。如果程序员是以索引字段搜寻数据，那么这些方法便会自动使用索引来搜寻数据。如果搜寻的字段不是索引字段，那么 TClientDataSet/TSimpleDataSet 也会根据情况使用最好的方法来搜寻数据。

本章讨论的内容就是如何使用这些方法搜寻数据，除了说明最常使用的搜寻方法之外，也会讨论在什么情况下应该使用什么方法来搜寻数据。更重要的是，本章在后半段会说明如何有效率地搜寻数据。虽然 TClientDataSet/TSimpleDataSet 提供的搜寻数据方法都非常好用，但是在许多情况下这些方法也可能非常没有效率。因此程序员必须知道如何能够有效率地搜寻数据。因此在本章中也说明了许多增加搜寻数据的性能的技巧，在了解了这些搜寻技巧之后，程序员就可以非常有信心地在各种结果数据集中有效率地搜寻数据了。

4.1 搜寻数据集数据

从本小节开始，本书将以实际的范例来说明如何搜寻结果数据集中的数据，并且使用范例 InterBase 数据库，你可以在本书的光盘中找到这个范例 InterBase 数据库。这个范例数据表是中文文化之后的 BIOLIFE 数据表，它的数据表纲要如图 4.1 所示。

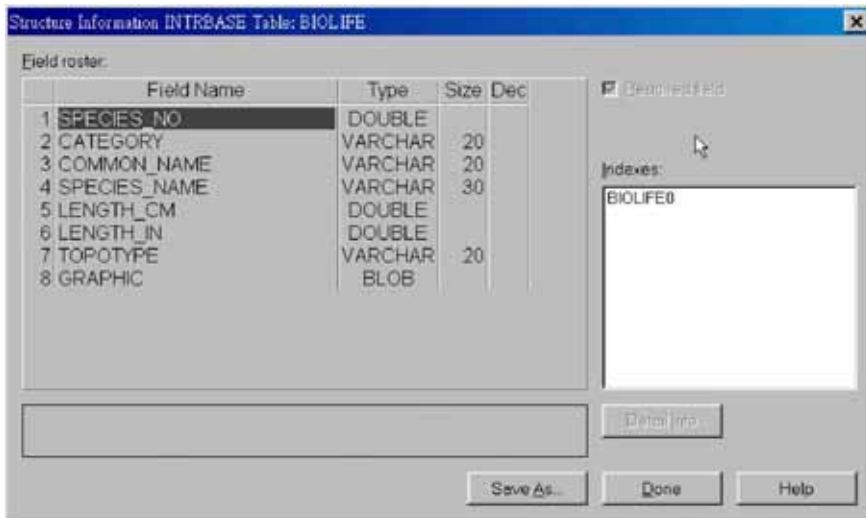


图4-1 本章使用的范例数据表纲要

首先在 Delphi 中建立一个新的项目，再点击 **File New Data Module** 建立一个数据模块。在数据模块中放入 **TSQLConnection** 组件，双击它激活组件编辑器以便将它连接到范例数据库 **CHINESEDEMO.GDB**。接着放入 **TSimpleDataSet**，设置它的 **DataSet\CommandText** 特性值为 `select * from BIOLIFE`，此时数据模块如图 4 2 所示，最后设置这个数据模块的名称为 **dmSearchData**。

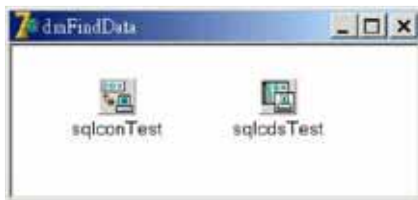


图4-2 范例应用程序使用的数据模块

现在回到主窗体，在上面放入 **TDataSouce** 并且将它连接到数据模块上的 **TSimpleDataSet**，再放入 **TDBNavigator**、**TDBGrid**、二个 **TEdit** 组件以及三个 **TButton** 组件。这三个 **TButton** 组件将在稍后使用不同的方法搜寻数据。最后在三个 **TButton** 组件的 **Caption** 特性值中输入 **Locate**、**Lookup** 和 **Filter**。此时主窗体如图 4 3 所示。

现在就让我们开始讨论如何使用 dbExpress 的搜寻方法。

4 1 1 Locate

TDataSet 组件以及它的派生组件（例如 **TSimpleDataSet**/**TClientDataSet** 等）都可以使用 **Locate** 方法在结果数据集中搜寻数据。程序首先必须使用 **SQL** 命令从后端数据

库中取得数据并且形成结果数据集，再使用 **Locate**方法搜寻数据。



图4-3 范例应用程序的主窗体

当使用 **Locate**方法搜寻数据时，程序员可以使用任何字段来搜寻，而不管这个字段是不是索引字段。当然，当程序员使用索引字段搜寻数据时，**Locate**会直接使用索引来帮助搜寻，因此速度会非常快。如果程序员使用非索引字段搜寻数据，那么 **Locate**也将使用目前它知道的最好的方式来搜寻数据。

此外，**Locate**方法不只能够搜寻单一的字段，它还能够同时用数个字段搜寻数据。程序员可以组合数个字段的搜寻条件在结果数据集中搜寻数据。

由于 **Locate**能够搜寻各种不同数据类型的字段，因此 **Locate**方法在设置搜寻条件时是以 **Variant**类型的变量来存储搜寻值。当程序员要使用多个字段搜寻数据时必须建立一个 **Variant**数组来存储搜寻值。

此外，**Locate**方法在搜寻数据时也能够使用模糊条件来寻找特定的数据，例如程序员可以要求 **Locate**在搜寻数据时不区分大小写，或是以部分字符串来搜寻数据，这为程序员提供了非常大的弹性。

下面就是 **Locate**的方法原型：

```
function Locate (const KeyFields:String; const KeyValues Variant; Options:
  TLocateOptions) : Boolean;
```

Locate方法接受三个参数，第一个参数 **KeyFields**是程序员要搜寻的字段名称。如果程序员要搜寻单一字段，那么只需要直接传入此字段名称。如果要按照多个字段

条件进行搜寻,那么程序员需要传入所有的字段名称,并且以分号分隔每一个字段名称。

第二个参数 **KeyValues**是指程序员欲搜寻的条件值。它的类型是 **Variant**,因为 **Variant**几乎可以代表任何类型,因此程序员可以搜寻整数、小数、字符串或是布尔值。同样,如果程序员只搜寻一个条件值,那么就可以直接在这个参数位置传入搜寻值。如果要按照多个字段条件进行搜寻,那么程序员必须建立一个 **Variant**数组,然后在这个数组中的每一个元素中指定条件值,再将这个 **Variant**数组传递到这个参数中。**Variant**数组可以使用 **VarArrayOf**方法或是使用 **VarArrayCreate**方法来建立,在稍后的范例中会有程序代码说明。

Locate方法的最后一个参数 **TLocateOptions**是让程序员在搜寻字符串字段时指定以什么标准来搜寻数据。程序员可以指明以不区分大小写的方式搜寻字符串数据,或是按照部分字符串值来搜寻数据。下面就是 **TLocateOptions**的类型定义:

```
type
    TLocateOption = (loCaseInsensitive, loPartialKey);
    TLocateOptions = setof TLocateOption;
```

在使用 **Locate**时,如果使用 **loCaseInsensitive**就代表不区分大小写,如果使用 **loPartialKey**就代表按照部分字符串搜寻数据。

Locate方法的返回值是一个布尔值,它代表 **Locate**方法是否成功地找到了要搜寻的数据。如果找到的话,就返回 **True**,否则就返回 **False**。当 **Locate**方法成功地搜寻到数据之后,它就会将目前的记录位置移动到这个记录上,否则就会停留在 **Locate**开始搜寻之前的记录位置上。

请注意, **Locate**方法搜寻数据的结果是一个记录,因此如果你想搜寻符合条件的多个记录,那么你可以使用稍后介绍的过滤器 (**Filter**) 功能。

现在让我们使用数个范例来说明如何使用 **Locate**方法。下面的范例程序代码以一个字段来搜寻数据,它是以数据表的 **NAME**字段来搜寻拥有“李维”这个值的记录,由于最后一个参数是空的,因此这代表 **NAME**字段必须拥有一模一样的“李维”这个值才算搜寻成功。

```
asSQLClientDataSet.Locate('NAME', 李维', []);
```

下面的程序代码则是以两个字段 **City**和**District**来搜寻数据,搜寻的目标是 **City**字段是“台北”而且**District**字段是“大安区”的记录。

```
asSQLClientDataSet.Locate('City;District', VarArrayQ[台北,大安区']), []);
```

下面的程序代码和第一个范例非常相像,只是这个程序代码搜寻的是 **NAME**字段以“李”开头的第一个记录。

```
asSQLClientDataSet.Locate('NAME', 李', [loPartialKey]);
```


最后一个范例则是搜寻 ID 字段以“A12”开头的第一个记录，而且不区分 A 的大小写。

```
asSQLClientDataSet.Locate('ID', 'A12', [loCaseInsensitive, loPartialKey]);
```

现在就让我们使用 **Locate** 方法在范例应用程序中搜寻数据。

1. 单字段搜寻

现在就让我们使用 **Locate** 方法在范例应用程序中搜寻数据，先让我们以单一字段来展示如何搜寻数据，稍后再说明如何以多个字段搜寻数据。现在请双击图 4 2 中的 **Locate** 按钮，并且在它的事件处理函数中编写如下的程序代码：

```
dmSearchData.sqlcdsTest.Locate('SPECIES_NO', editID.Text, [loCaseInsensitive,  
loPartialKey]);
```

这行程序代码使用数据模块中的 **TSimpleDataSet** 在 **SPECIES_NO** 字段中搜寻用户在 **TEdit** 控件 **editID** 中输入的数值。现在请执行这个范例应用程序，并且在主窗体中右边的 **TEdit** 控件中输入数值来搜寻数据。例如，图 4 4 便是范例应用程序执行的画面。当笔者在 **TEdit** 控件中输入 90100 并且点击 **Locate** 按钮之后，**TSimpleDataSet** 便会立刻找到这个记录并且把目前的记录位置移动到这个记录上。

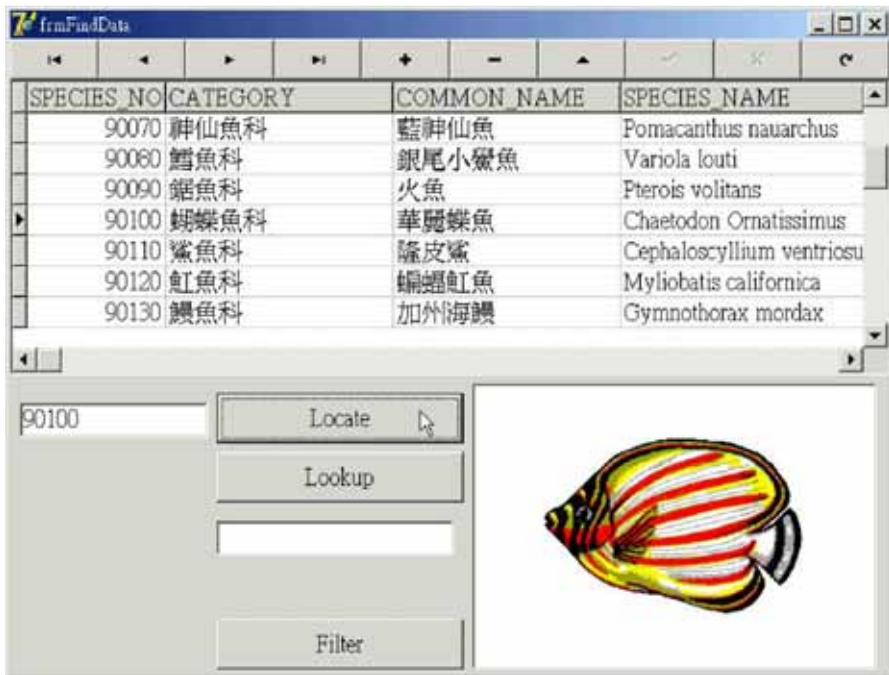


图4-4 Locate找到90100这个记录

使用 **Locate** 方法搜寻单一字段的数据是非常简单的，现在再让我们看看如何使用多个字段来搜寻数据。

2. 多字段搜寻

在Delphi/Kylix中建立一个应用程序，再和刚才的范例一样建立一个数据模块，并且放入 TSQLConnection和 TSimpleDataSet，连接到相同的范例数据库 CHINESEDEMO.GDB。接着在主窗体中放入图 4 5所示的控件。在主窗体中我们使用了一个 TComboBox，在这个 TComboBox中将会填入范例数据表的所有字段名称以便让用户可以自由选择要用来搜寻的字段。

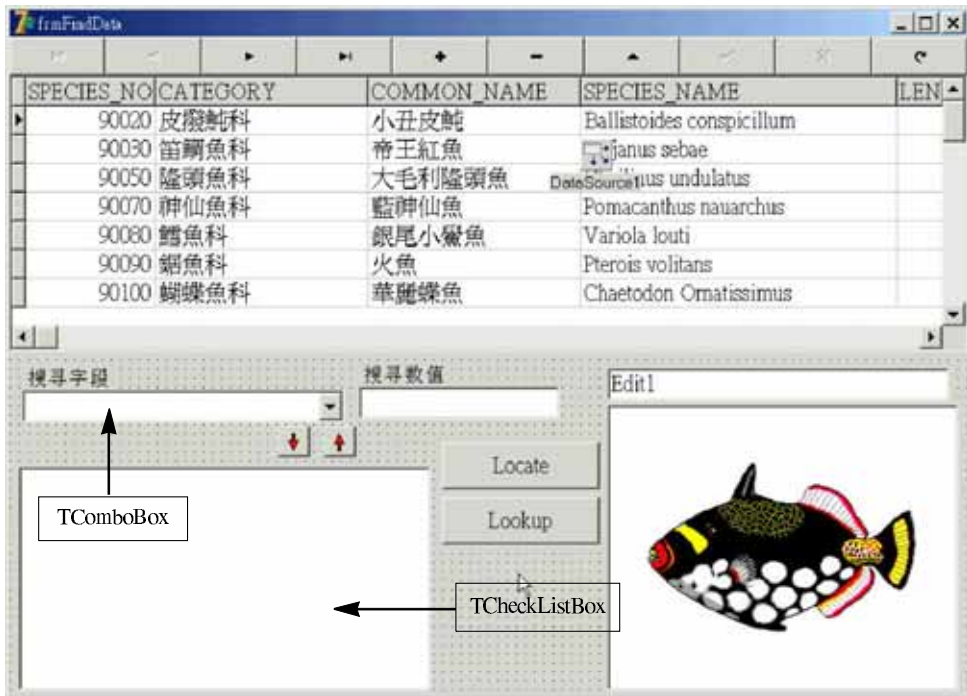


图4-5 范例应用程序的主窗体

主窗体另外使用了一个 TCheckBox，它主要用来存储用户输入的所有搜寻条件。用户在搜寻字段中选择一个字段，然后在搜寻值控件中输入欲搜寻的数值，那么就可以点击主窗体中向下箭头按钮把这个搜寻条件加入到 TCheckBox中。当用户输入完所有搜寻条件之后，就可以点击主窗体中的 Locate按钮开始搜寻数据。此时范例应用程序就会分析 TCheckBox中的所有搜寻字段和搜寻数值，再调用 Locate按照多个字段条件来搜寻数据。

图4 6就是这个范例应用程序执行的画面。当范例应用程序执行后，用户可以在 TComboBox中选择欲搜寻的字段，接着可以在搜寻值控件中输入欲搜寻的数值，接着点击向下箭头按钮将搜寻条件加入到 TCheckBox中，或是点击向上箭头按钮清除某一个搜寻条件。

在TCheckBox中的搜寻条件是以：

搜寻字段名称\搜寻字段值

的格式存储的。当用户点击了 **Locate** 按钮之后，就会从 **TCheckListBox** 中一一取出搜寻条件并且分析出搜寻字段名称以及搜寻字段值，再放入到 **Locate** 方法的第一个和第二个参数中。



图4-6 执行范例应用程序的画面

最后，当输入完所有的搜寻条件之后，用户就可以点击主窗体中的 **Locate** 按钮来搜寻数据了。例如，图 4 7 便是搜寻 **TOPOTYPE** 字段包含“台湾沿海”而且 **SPECIES NAME** 字段以字母 O 开头的记录。在点击了 **Locate** 按钮之后，范例应用程序调用 **Locate** 方法并且以多个字段为搜寻条件，果然立刻找到了这个记录。

这个范例应用程序是如何运作的呢？这个范例应用程序基本上执行了下列的工作：

- 1) 在程序激活时在 **TComboBox** 中填入范例数据表中的所有字段名称。
- 2) 在点击向下箭头按钮时把搜寻字段和搜寻值加入到 **TCheckListBox** 中，以及在点击向上箭头按钮时清除搜寻条件。
- 3) 在点击 **Locate** 按钮时从 **TCheckListBox** 中取出搜寻条件并且填入 **Locate** 方法的参数中，搜寻数据。

现在就让我们实现以上的工作。首先在范例应用程序激活时，访问数据模块中 **TSimpleDataSet** 的 **Field** 对象的 **FieldName** 特性值以取得字段名称，再填入 **TComboBox** 中：

```

procedure TForm1.FormActivate (Sender: TObject;
var
    iField : Integer;
begin
    for iField := 0 to dmFindData.sqlcdsTest.FieldCount - 1 do
    begin
        cbFields.Items.Add (dmSearchData.sqlcdsTest.Fields[iField].FieldName)
    end;
    cbFields.ItemIndex := 0;
end;

```

当点击向下箭头按钮时，取出用户在 TComboBox 中选择的字段名称以及在搜寻值控件中输入的搜寻值，再检查 TCheckListBox 中是否已经存在了这个搜寻字段。如果没有的话，就把字段名称和字段值加入到 TCheckListBox 中。



图4-7 以数个字段条件来搜寻数据

此外当点击向上箭头按钮时，我们就删除 TCheckListBox 中目前被选择的搜寻条件。

```

procedure TForm1.sbbtnAddClick (Sender: TObject;
begin
    if (not AlreadyInCond (cbFields.Text)) then
    begin
        clbConditions.Items.Add (cbFields.Text + '\' + lblSearchValue)Text
    end;

```

```

        clbConditions.Checked[clbConditions.Count - 1] := True;
    end;
end;

```

```

procedure TForm1.sbbtnDeleteClick(Sender: TObject);
begin
    try
        clbConditions.Items.Delete(clbConditions.ItemIndex);
    except
        on Exception do;
    end;
end;

```

最后当用户点击了主窗体中的 **Locate** 按钮时，范例应用程序就先检查用户是否输入了任何搜寻条件。如果有的话，就调用 **GetSerchFields** 从 **TCheckListBox** 中取出所有搜寻字段名称，再调用 **GetSearchValues** 取得用户输入的所有搜寻值，最后调用 **Locate** 方法来搜寻数据。

其中的 **GetSearchValues** 会先调用 **VarArrayCreate** 方法建立一个 **Variant** 数组，再在这个 **Variant** 数组中一一输入用户指定的搜寻值。

```

procedure TForm1.btnLocateClick(Sender: TObject);
var
    sFields : String;
begin
    lStart := GetTickCount;
    if (CanSearch) then
        begin
            sFields := GetSerchFields;
            dmFindData.sqlcdsTest.Locate(sFields, GetSearchValues[loCaseInsensitive,
                loPartialKey]);
        end;
    lEnd := GetTickCount;

    Self.Caption := FloatToStr((lEnd - lStart)/1000.0);
end;

function TForm1.GetSearchValues : Variant;
var
    iCount : Integer;
    sCond : String;
begin
    Result := VarArrayCreate([0, Self.clbConditions.Items.Count - 1],
        varVariant);

```

```

    for iCount := 0 to Self.clbConditions.Items.Count - 1 do
    begin
        sCond := Self.clbConditions.Items[iCount];
        Result[iCount] := GetSearchValue(sCond);
    end;
end;

function TForm1.GetSearchFields: String;
var
    iCount : Integer;
    sCond : String;
begin
    Result := '';
    for iCount := 0 to Self.clbConditions.Items.Count - 1 do
    begin
        sCond := Self.clbConditions.Items[iCount];
        Result := Result + GetSearchField(sCond) + ';';
    end;
    Delete(Result, Length(Result), 1);
end;

function TForm1.CanSearch: Boolean;
begin
    Result := Self.clbConditions.Items.Count > 0;
end;

function TForm1.GetSearchField(const sCond: String): String;
var
    iPos : Integer;
begin
    iPos := Pos('\', sCond);
    Result := Copy(sCond, 1, iPos - 1);
end;

function TForm1.GetSearchValue(const sCond: String): String;
var
    iPos : Integer;
begin
    iPos := Pos('\', sCond);
    Result := Copy(sCond, iPos + 1, Length(sCond) - iPos);
end;

```

上面的范例展示了如何使用 **Locate**方法按照多个字段条件来搜寻数据。由于 **Locate**方法的第二个参数是 **Variant**类型的，因此我们可以搜寻几乎任何类型的字段。

Locate方法非常适合在所有数据已经存在于结果数据集中的应用程序使用，但是对于拥有大量记录的数据表而言却不见得适合，在稍后的小节中本章会继续讨论如何使用Locate在大量数据中搜寻数据。

4.1.2 Lookup

Lookup方法和上一小节介绍的 Locate方法在使用上非常类似，它们的差别是当 Locate找到要搜寻的数据后，它会把目前的记录位置移动到找到的这个记录上；但是当Lookup找到要搜寻的数据后，它会返回找到的记录的特定字段值，却不会移动目前的记录位置。下面即是Lookup方法的原型：

```
function Lookup (const KeyFields: String; const KeyValues: Variant; const  
ResultFields: String): Variant;
```

Lookup方法的第一个参数也是用户欲搜寻的字段名称，每一个欲搜寻的字段也是使用分号分隔。第二个参数则是欲搜寻的字段值，如果欲搜寻多个字段，那么这个参数可以是 Variant数组。Lookup方法的第一和第二个参数的意义和前面 Locate方法是一样的。

Lookup方法的第三个参数则是指定当 Lookup找到欲搜寻的数据之后，要返回这个记录的哪些字段值。如果程序员想让 Lookup返回多个字段值，那么每一个字段也是以分号分隔。

Lookup方法返回的数值就是第三个参数指定的字段值。如果 Lookup返回多个字段的话，那么这个返回值就是一个 Variant数组，每一个返回的字段存储在这个 Variant数组的元素中。

现在就让我们看看如何使用 Lookup方法搜寻数据，首先先以简单的单一字段来搜寻数据。

1. 单字段搜寻

请使用鼠标双击图 4.5中的Lookup按钮，并且在它的事件处理函数中编写如下的程序代码：

```
try  
  edtReturn.Text :=  
    dmFindData.sqlcdsTest.Lookup ('SPECIES_NO', edtID.Text, 'COMMON_NAME');  
except  
  on e : Exception do  
    ShowMessage (E.Message);  
end;
```

上面的程序代码用Lookup方法搜寻 BIOLIFE数据表中的SPECIES_NO字段，并且以用户在TEdit控件edtID中输入的数值作为搜寻的目标，要求Lookup方法在搜寻到数据之后返回COMMON_NAME字段的值，并且把返回值显示在TEdit控件edtReturn中。

图4 8便是执行范例应用程序并且输入 90100搜寻数据的画面。从画面中可以看到当Lookup找到90100这个记录之后，它果然会返回 COMMON NAME字段的值“华丽蝶鱼”，但是目前记录的位置仍然停留在原先的记录上，而不会移动到 90100这个记录上。

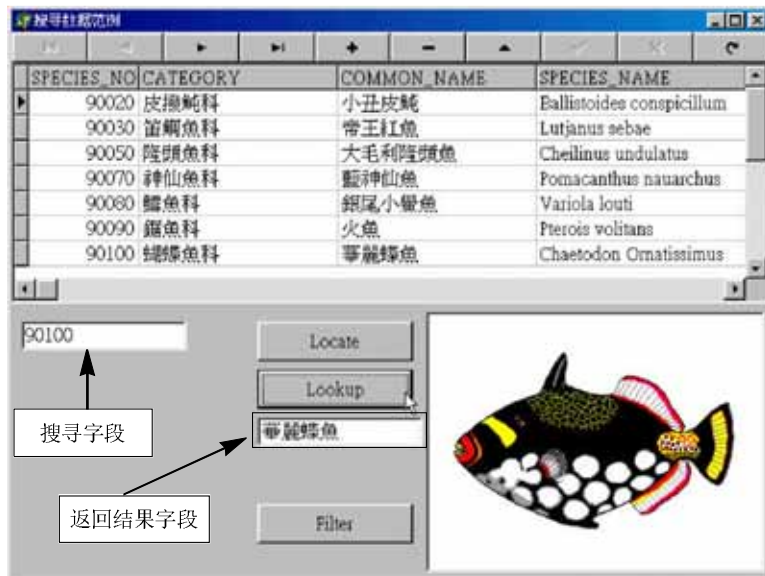


图4-8 执行范例应用程序的画面

使用Lookup搜寻并且返回单一字段的数据是非常简单的，再让我们看看如何搜寻和返回多个字段的数据。

2. 多字段搜寻

请将上一小节的Lookup按钮的事件处理函数修改为如下的程序代码：

```
procedure TForm1.btnLookupClick(Sender: TObject);
var
  sFields : String;
  vResult : Variant;
  iCount : Integer;
begin
  lStart := GetTickCount;
  if (CanSearch) then
  begin
    sFields := GetSearchFields;

    vResult:=dmFindData.sqlcdsTest.Lookup(sFields,GetSearchValues,edtResult.Text);

    if (VarIsArray(vResult)) then
    begin
```

```
sFields := '';

for iCount:=VarArrayLowBound(vResult,1)to VarArrayHighBound(vResult,1) do
begin
    sFields := sFields + ';' + vResult[iCount];
end;
Delete(sFields, 1, 1);
edtResult.Text := sFields;
end
else
    edtResult.Text := vResult;
end;
lEnd := GetTickCount;

Self.Caption := FloatToStr(( lEnd - lStart )/1000.0);
end;
```

上面的程序代码先调用 **GetSearchValues** 从主窗体中的 **TCheckListBox** 中取得用户欲搜寻的所有字段和值，再从 **edtResult** 中取得用户欲取得的字段返回值，然后调用 **Lookup** 方法。由于用户可以在 **edtResult** 中输入多个字段，因此当 **Lookup** 执行完毕之后，我们先检查返回的是不是一个 **Variant** 数组。如果是的话，就进入一个循环，调用 **VarArrayLowBound** 以及 **VarArrayHighBound** 取得数组中实际的元素值范围，然后再一一从数组元素中取出 **Lookup** 返回的字段值，最后再把这些返回的数值显示在 **edtResult** 控件中。

图4 9到图4 11便是执行范例应用程序的画面，图4 9是用新修改过的程序代码搜寻并且返回一个字段数据的画面。而图4 10是在 **TOPOTYPE** 字段中搜寻“台湾沿海”，并且要求返回 **COMMON NAME** 和 **CATEGORY** 两个字段的值。图4 11则是点击了 **Lookup** 按钮之后看到应用程序果然返回了 **COMMON NAME** 和 **CATEGORY** 这两个字段的值。

请注意，**Lookup** 并不像 **Locate** 那样可以使用模糊搜寻，**Lookup** 只能搜寻到完全一样的字段值。因此如果读者使用部分字符串来搜寻数据的话，那么 **Lookup** 不会找到数据，而会产生一个搜寻错误的异常。

4 1 3 过滤器

除了 **Locate** 和 **Lookup** 之外，事实上 **Delphi/Kylix** 的过滤器（**Filter**）功能是非常有用的，因为过滤器不但能够搜寻数据，更棒的是它可以再把结果数据集中的数据分门别类，让应用程序只访问特定的数据组。这个行为相当于对从后端数据库返回的结果数据集中的数据再使用额外的条件，以此取得子结果数据集。

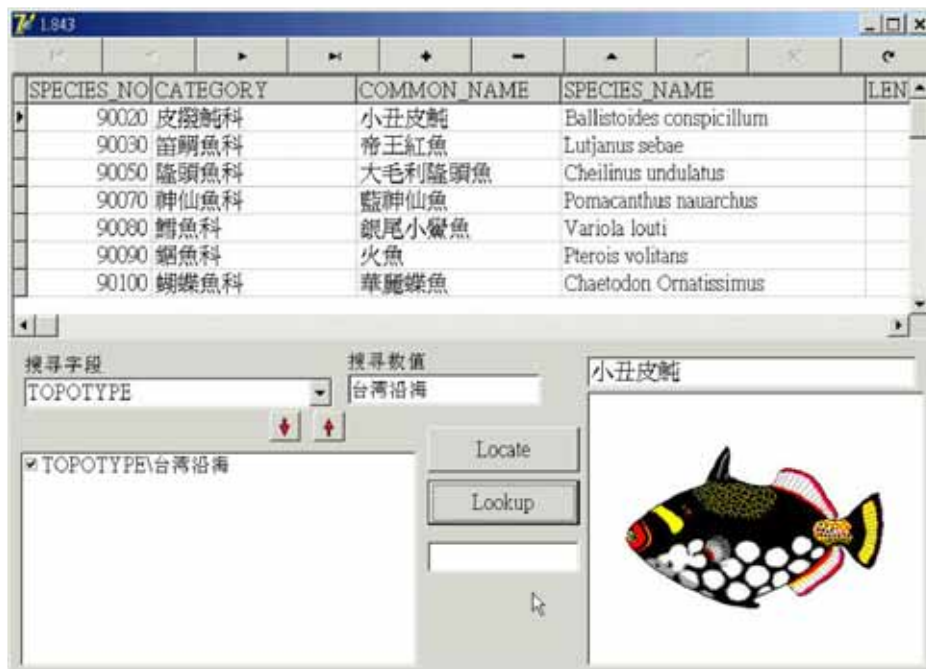


图4-9 以Lookup方法搜寻一个字段并且返回一个字段的数据

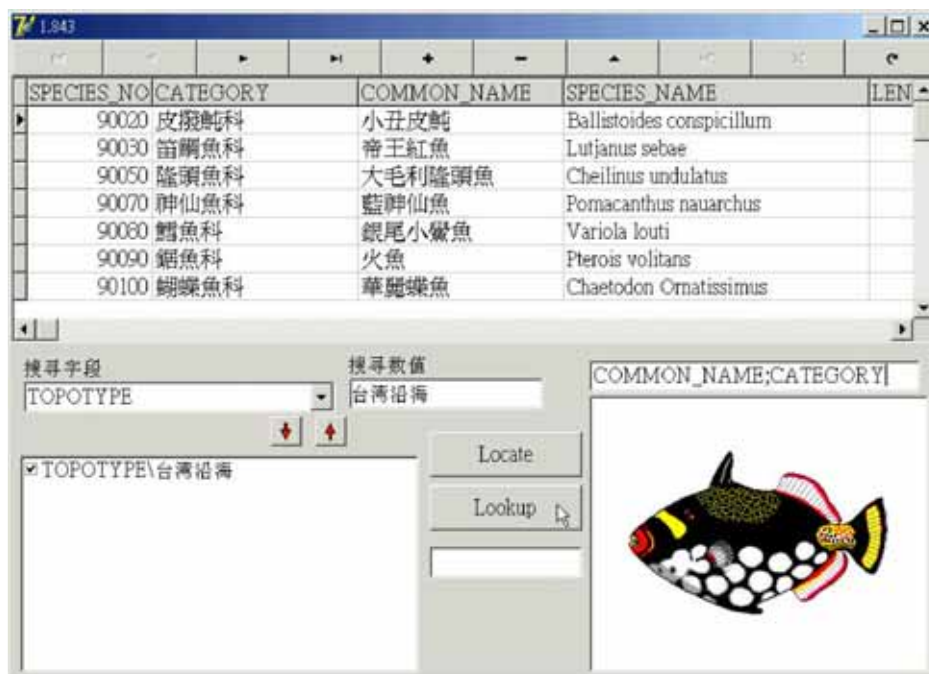


图4-10 以Lookup方法搜寻一个字段并且返回数个字段的数据

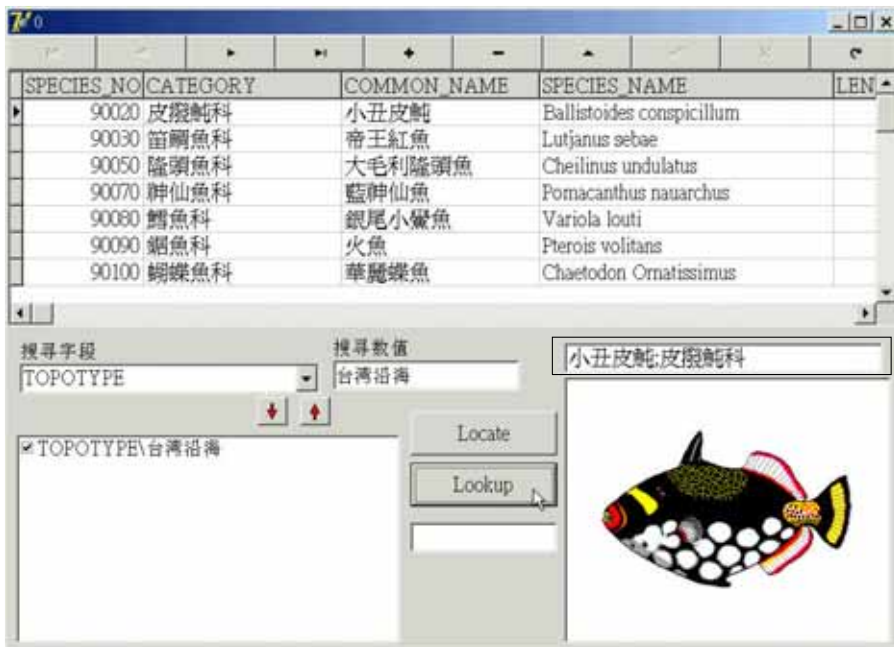


图4-11 以Lookup方法搜寻一个字段并且返回数个字段的数据

Delphi/Kylix的过滤器功能使程序员不但可以对单一字段设置过滤条件，也可以对多个字段设置过滤条件，同时不限定字段是否是索引字段。此外，程序员有两种使用过滤器的方法，第一个方法是使用 TSimpleDataSet/TClientDataSet组件的Filter特性值，第二个方法是使用 TSimpleDataSet/TClientDataSet组件的 OnFilterRecord事件处理函数。

这两种方法的差异是 Filter特性值只能让程序员使用字符串来设置过滤条件，而 OnFilterRecord事件处理函数却能够让程序员使用 Object Pascal来执行任何复杂的过滤条件。Delphi/Kylix提供的与过滤功能相关的特性和事件处理函数如下表所示：

特性/事件处理函数	意 义
Filter特性值	使用字符串条件来过滤数据
OnFilterRecord事件处理函数	使用事件处理函数程序代码来过滤数据
Filtered	决定是否开启过滤器功能的特性值
FilterOptions	过滤功能使用的额外条件

要想使用过滤器功能，程序员必须将 Filtered特性值设置为 True。一旦 Filtered特性值设置为 True之后，如果 Filter特性值中有任何过滤条件，Delphi/Kylix便会以这个过滤条件做为标准来过滤目前在结果数据集中的数据，只有符合过滤条件的数据才能够在数据感知组件中显示或是被访问，而不符合过滤条件的数据暂时无法被访问。此外，如果程序员定义了 OnFilterRecord事件处理函数，那么 Delphi/Kylix也会执行

OnFilterRecord事件处理函数来过滤数据。因此如果程序员同时设置了 Filter特性值以及OnFilterRecord事件处理函数,那么这两个过滤条件都会被执行。

FilterOptions特性则类似于Locate方法的第3个参数,用来指明在过滤数据时是否需要区分大小写,或者是否需要对比完整的字符串过滤值。

现在就让我们看看如何使用过滤器来过滤数据,首先修改图 4 5的主窗体,在主窗体中加入另外三个按钮, Filter、OnFilter和SetRange(如图4 12所示),以便在范例应用程序中使用过滤器和SetRange方法搜寻数据。

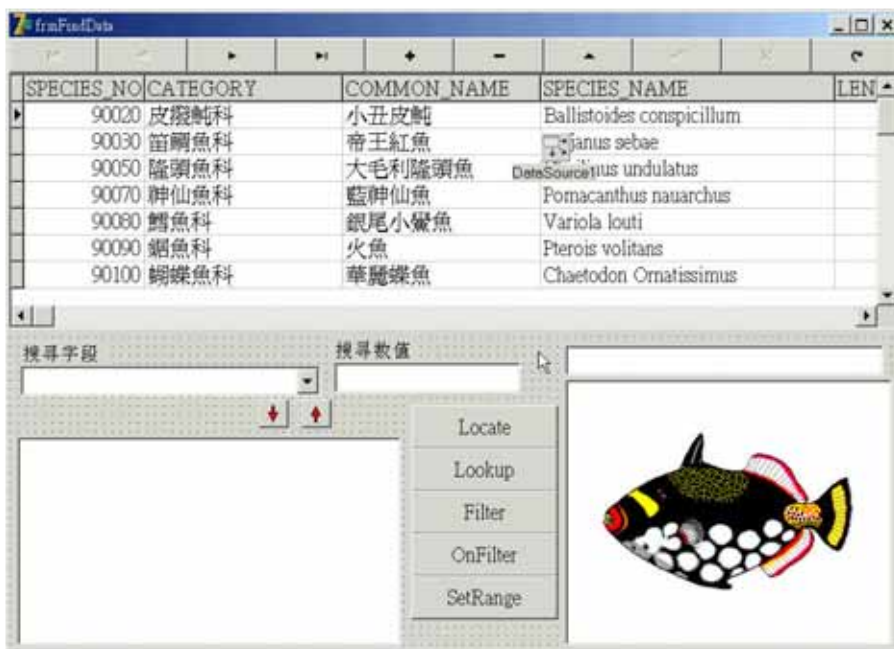


图4-12 在主窗体中加入Filter、OnFilter和SetRange按钮

现在我们要在范例应用程序中使用过滤器功能来搜寻数据,请双击主窗体中的Filter按钮,并且在它的事件处理函数中编写如下的程序代码:

```
procedure TForm1.btnFilterClick(Sender: TObject);
begin
    dmFindData.sqlcdsTest.Filtered := False;
    dmFindData.sqlcdsTest.Filter := edtResult.Text;
    dmFindData.sqlcdsTest.Filtered := True;
end;
```

上面的程序代码首先关闭过滤器功能,因为现在我们要改变过滤条件,接着把用户在edtResult控件中输入的过滤条件设置给 TSimpleDataSet的Filter特性值。最后再设置TSimpleDataSet的Filtered特性值为True以便开启过滤器功能,并且使用新的过滤条件来搜寻(过滤)数据。

图4 13是执行范例应用程序，并且在 `edtResult`控件中输入下面的过滤条件：

```
LENGTH_CM>=60 AND LENGTH_CM <= 80
```

接着点击主窗体中的 **Filter**按钮，那么我们就可以在数据感知组件中看到现在只有符合这个过滤条件的数据才会显示出来。



图4-13 执行范例应用程序并且使用过滤器来搜寻数据

请注意，现在我们设置过滤条件是以 `LENGTH CM`字段为目标的，而 `LENGTH CM`字段并不是索引字段，但是 `Delphi/Kylix`的过滤器功能仍然能够找到数据。

当然过滤器并不是只能以一个字段作为过滤目标，程序员可以同时使用数个字段来过滤数据，例如下面的过滤条件同时以 `LENGTH CM`和 `TOPOTYPE`这两个字段作为目标来搜寻数据。请注意由于 `TOPOTYPE`字段是字符串类型的，因此它的搜寻值必须使用单引号来设置。

```
LENGTH_CM>=10 AND LENGTH_CM <= 80 AND TOPOTYPE=' 台湾沿海 '
```

图4 14便是执行这个过滤条件后的画面，从画面中我们看到范例应用程序仍然找到了符合过滤条件的数据。

`Delphi/Kylix`的过滤器功能非常强大，它几乎没有什么限制。不过如果读者觉得使用字符串值来设置过滤条件仍然不够，那么还可以使用 `OnFilterRecord`事件处理函数来建立过滤条件。

1. 使用 `OnFilterRecord`事件处理函数

`TClientDataSet/TSimpleDataSet`组件有一个 `OnFilterRecord`事件处理函数，可以让

程序员使用 Object Pascal 程序代码设置任何复杂的过滤条件。OnFilterRecord 事件处理函数的原型如下：

```
procedure FilterRecord (DataSet: TDataSet; var Accept: Boolean);
```

当程序员开启过滤器功能时，Delphi/Kylix 会为结果数据集中的每个记录调用一次 OnFilterRecord 事件处理函数。OnFilterRecord 事件处理函数的第一个参数是目前执行过滤功能的数据集组件，而第二个参数 Accept 是一个 var 类型的布尔值参数。如果在 OnFilterRecord 事件处理函数中这个记录符合过滤条件的话，那么程序员必须设置 Accept 为 True，以便让这个记录存在于经过过滤之后的子结果数据集中。相反，如果这个记录不符合过滤条件，那么程序员就必须设置 Accept 参数为 False 以剔除这个记录。

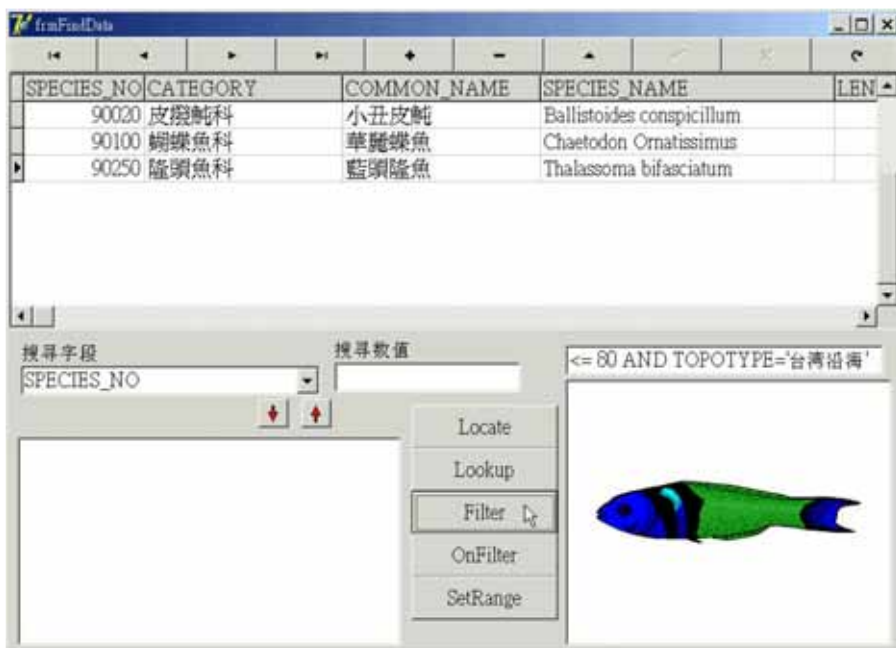


图4-14 执行范例应用程序并且使用过滤器来搜寻数据

现在就让我们在范例应用程序中使用 OnFilterRecord 事件处理函数来过滤数据。在刚才使用过滤器的范例中，我们以 LENGTH CM 和 TOPOTYPE 这两个字段为目标来搜寻数据。如果现在我们希望寻找 LENGTH CM 在 10 和 50 之间并且 TOPOTYPE 以‘台湾’开头的记录，那么使用 OnFilterRecord 事件处理函数可以非常简单地找到我们需要的数据。

现在请双击图 4 12 中的 OnFilter 按钮，并且在它的 OnClick 事件处理函数中编写如下程序代码：

```
procedure TForm1.btnOnFilterRecordClick (Sender: TObject);
```

```
begin
    dmFindData.sqlcdsTest.Filtered := False;
    dmFindData.sqlcdsTest.Filter := '';
    dmFindData.sqlcdsTest.Filtered := True;
end;
```

在上面的程序代码中，我们先关闭过滤器功能并且清除 `TSimpleDataSet` 的 `Filter` 特性值以避免 Delphi/Kylix 同时使用 `Filter` 特性值和 `OnFilterRecord` 事件处理函数过滤数据。

接着在 `sqlcdsTest` 的 `OnFilterRecord` 事件处理函数中编写如下的程序代码：

```
procedure TdmSearchData.sqlcdsTestFilterRecord (DataSet: TDataSet;
    var Accept: Boolean);
begin
    if ( (DataSet.FieldName('LENGTH_CM').Value >= 10) and
        (DataSet.FieldName('LENGTH_CM').Value <= 50) and
        ( Pos('台湾', DataSet.FieldName('TOPOTYPE').Value) <> 0 ) ) then
        Accept := True
    else
        Accept := False;
end;
```

在上面的程序代码中，先使用第一个参数 `DataSet` 来找到目前正在被过滤的数据，判断它的 `LENGTH_CM` 字段值是否在 10 和 50 之间，而且 `TOPOTYPE` 字段值是不是以‘台湾’开头的，如果是的话就设置 `Accept` 为 `True`，反之则设置 `Accept` 为 `False`。

现在执行范例应用程序并且点击主窗体中的 `OnFilter` 按钮，那么我们就可以看到如图 4 15 所示的画面。

范例应用程序果然顺利地找到了符合过滤条件的数据。请注意，由于过滤器功能对结果数据集中的每一个记录都进行一次过滤操作，因此如果结果数据集中的数据量很大的话，那么在使用过滤器功能之前程序员必须很小心地评估过滤器的影响。

2. 使用过滤器的场合

虽然过滤器可以像 `TClientDataSet`/`TSimpleDataSet` 的 `Locate` 和 `Lookup` 方法一样搜寻数据，但是过滤器另外有一个非常特别的用途，那就是它可以在结果数据集中再用过滤条件取得子结果数据集。这个用途在一些应用中是非常方便的，让我们使用一个例子和图 4 16 来说明这个用途。

假设在我们的应用程序中已经使用 `SQL` 命令从后端数据源中获取了数量不多的数据到结果数据集中，例如笔者使用 `SQL` 命令搜寻在台北市使用 Delphi/Kylix 而且购买了本书的程序员。那么如果笔者又想根据这些读者所在的行政区来分析购买的情况，例如以大安区、文山区和中山区等区域，那么笔者仍然可以再使用 `SQL` 命令来访问数据。但是既然这些数据已经存在于结果数据集中，那么笔者可以直接使用过滤器

的功能来取得数据，而不需要再使用额外的 SQL 命令从后端数据源中重新获取数据，这种数据处理方式的性能比重新使用 SQL 命令好。



图4-15 执行范例应用程序并且使用 OnFilterRecord 事件处理函数来搜寻数据

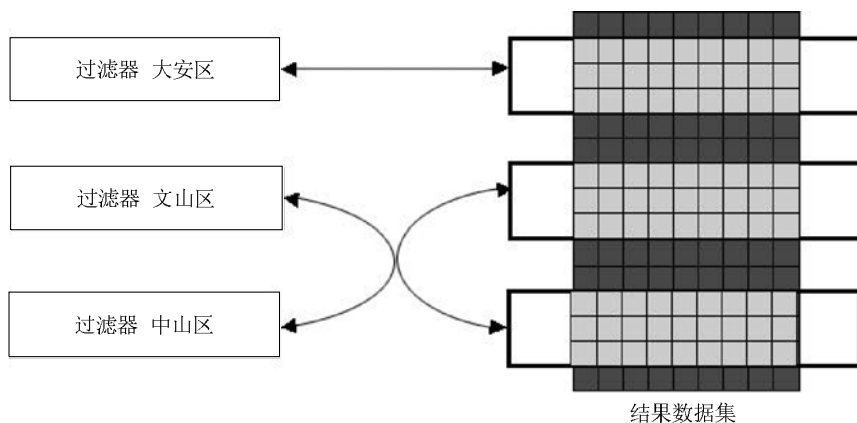


图4-16 使用过滤器在结果数据集中过滤出子结果数据集

在笔者平常编写数据库应用程序时，也经常使用这个技巧从结果数据集中访问笔者需要的数据。由于过滤器的性能在结果数据集中数据不多时是不错的，因此程序员也可以善用这个技巧来处理数据。

4.1.4 Range

本章要说明的最后一个搜寻方法就是 **SetRange**。**SetRange**方法让程序员可以指定索引字段符合一定范围值的数据才可以被访问。程序员可以通过 **SetRange**设置特定的索引字段值来搜寻一个记录，或是一组在指定范围值之内的记录。由于 **SetRange**只能用在索引字段上，因此它的适用性比前面介绍的搜寻方法小，不过 **SetRange**在搜寻索引字段的数据时会比其他方法效率高一点点。

使用**SetRange**方法是非常简单的，程序员只需要指定索引范围值的起始值和结尾值，**Delphi/Kylix**便会自动地从结果数据集中搜寻出所有符合范围值的记录。下面是**SetRange**的函数原型：

```
procedure SetRange(const StartValues, EndValues:array of const);
```

SetRange的第一个参数 **StartValues**是搜寻范围的起始值，而 **EndValues**参数是搜寻范围的结尾值，使用起来非常简单。现在就让我们在范例应用程序中使用**SetRange**搜寻数据。

在主窗体中再加入一个**SetRange**按钮以及两个**TEdit**控件以便输入搜寻范围的起始值和结尾值，如图4-17所示。



图4-17 加入SetRange的范例应用程序主窗体

双击主窗体中的**SetRange**按钮并且在OnClick事件处理函数中编写如下的程序代码：

```

procedure TForm1.btnSetRangeClick(Sender: TObject);
begin
    lStart := GetTickCount;
    dmFindData.sqlldsTest.SetRange([edtStartRange.Text], [edtEndRange.Text]);
    lEnd := GetTickCount;

    Self.Caption := FloatToStr((lEnd - lStart)/1000.0);
end;

```

上面的程序代码以 `edtStartRange` 和 `edtEndRange` 控件的输入值作为 `SetRange` 的搜寻范围的起始值和结尾值，并且以 `constant` 类型的数组作为调用 `SetRange` 方法的参数。

图4 18便是执行范例应用程序并且点击 `SetRange` 按钮后的结果，从画面中可以看到 `SetRange` 方法果然快速地搜寻到了在搜寻范围值之内的所有数据。

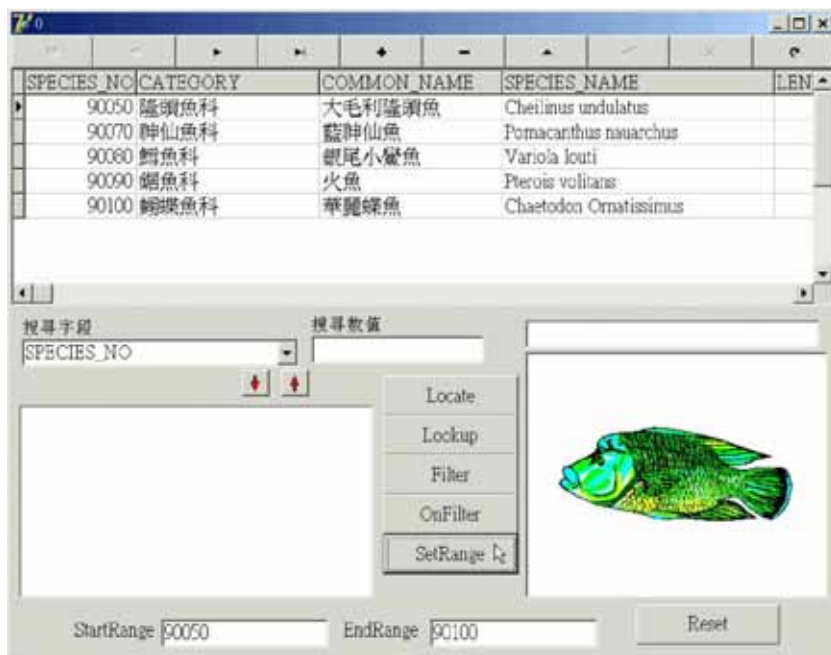


图4-18 使用 `SetRange` 方法搜寻符合范围值的数据

当程序员处理完由 `SetRange` 获取的数据之后，可以调用 `CancelRange` 取消 `SetRange` 的作用，而恢复到结果数据集中所有数据都可以被访问的状态。

```

procedure CancelRange;

```

4.2 搜寻方法的比较

前面的小节讨论了许多在结果数据集中搜寻数据的方法，本小节把这些方法的

使用特性整理出来，而且提供一些规则让程序员能够根据需求选择最适合的搜寻方法。

下面的表格总结了前面四种搜寻方法在搜寻数据时是否能够搜寻到多个记录。从下面的表格中可以看到，如果程序员希望在结果数据集中搜寻到符合条件的多个记录，那么可以使用过滤器或是 **SetRange** 方法，而 **Locate** 和 **Lookup** 方法则只适合搜寻单个记录。

	搜寻单个记录	搜寻多个记录
Locate	●	
Lookup	●	
过滤器	●	●
SetRange	●	●

下面的表格则列出了四种方法能够搜寻的字段类型。这四种搜寻方法都可以搜寻索引字段，但是对于非索引字段来说，**SetRange** 是无法胜任的。

	搜寻非索引字段	搜寻索引字段
Locate	●	●
Lookup	●	●
过滤器	●	●
SetRange		●

性能比较

虽然前面介绍的四种搜寻方法在适用场合方面各有不同，但是这些方法也可以执行类似的搜寻工作。因此如果在一个搜寻工作中同时可以使用这四种方法，那么除了可能因为程序员个人的喜好不同而选择不同的方法之外，到底哪一个方法的执行性能会比较好？还是这些方法的性能结果都是一样的？这些方法搜寻索引字段的结果和搜寻非索引字段的结果是不是一样？搜寻少量数据和搜寻大量数据有没有什么不同？

这些问题也许许多读者不曾注意的，但是在结果数据集中搜寻数据是许多应用程序经常执行的工作，因此了解并且能够应付这些问题是非常重要的。本小节就会讨论许多重要的问题，并且在稍后的章节中继续说明如何有效率地搜寻数据。

1. 搜寻索引字段

当程序员使用 **Locate**、**Lookup**、过滤器和 **SetRange** 搜寻索引字段的数据时，到底它们的性能如何？图 4-19 是使用这些方法在 **CHINESEDEMO.GDB** 中搜寻索引字段数据的结果。从下表中可以看出，这些方法的性能几乎都是一样的，因此使用哪一种方法搜寻数据并没有多大的差异。

方 法	费时（秒）
Locate	0.077
Lookup	0.078
Filter	0.079
OnFilterRecord	0.080
SetRange	0.078

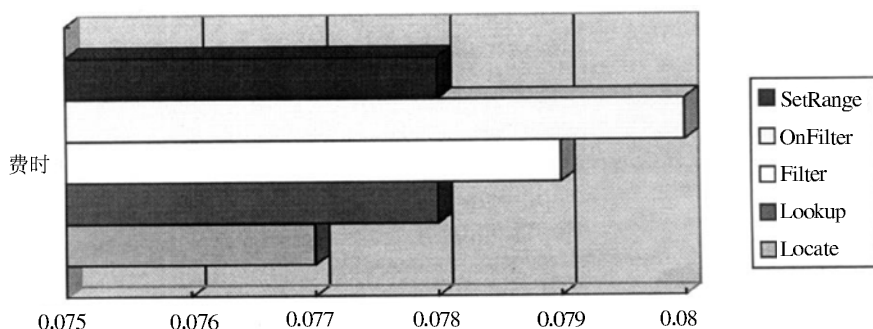


图4-19 搜索索引字段的执行结果

2. 搜寻非索引字段

下面的表格和图 4 20则是搜寻非索引字段的执行结果，令人讶异的是搜寻非索引字段和搜索索引字段在性能上并没有什么不同。

方 法	费时（秒）
Locate	0.078
Lookup	0.078
Filter	0.079
OnFilterRecord	0.080
SetRange	NA

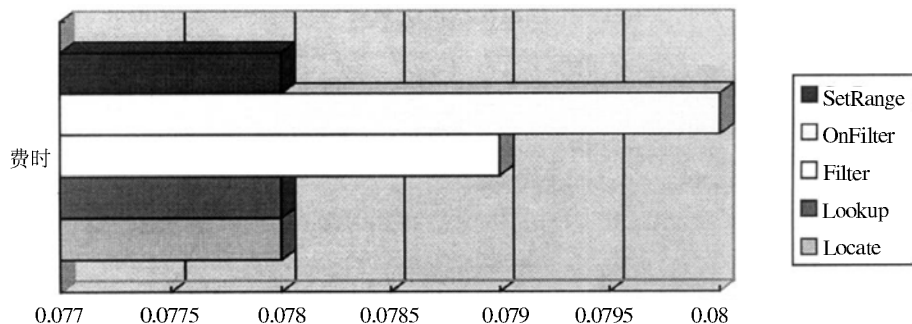


图4-20 搜寻非索引字段的执行结果

3. 搜寻拥有大量数据的数据表

接着我在一个拥有 50000个记录的数据表中搜寻一个特定的记录，下面的表格和

图4 21显示了执行的结果：

方 法	费时（秒）
Locate	11.988
Lookup	12.073
Filter	12.21
OnFilterRecord	12.875
SetRange	11.687

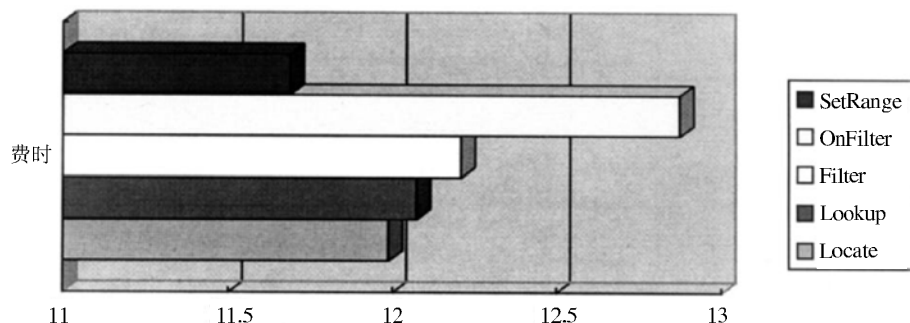


图4-21 搜寻大型结果数据集的结果

从这个结果中可以看到SetRange有最好的性能，但是对于搜寻一个记录需要超过10秒的结果来说，快了零点几秒并没有什么意义，都是一样的缓慢。因此这些搜寻方法对于结果数据集中拥有大量数据的情况并不适用，稍后的章节会说明如何克服这种状况。

从上面的各种执行数据来看，当结果数据集中的数据量并不多时，例如 CHI NESEDEMO.GDB只有几十个记录，那么Locate、Lookup、Filter和SetRange都非常有效率，OnFilterRecord事件处理函数似乎表现得更好。而在少量数据的结果数据集中搜索索引字段和非索引字段似乎也没有多大的分别，因为就如Delphi/Kylix联机帮助中说明的一样，这些方法会自动使用最好的方式来搜寻数据。但是请注意，这个结果是针对数据表中数据量少的情况，如果数据表中包含大量的数据，那么不管读者使用哪一种搜寻方法都是非常缓慢的，在稍后的章节中本书会详细地说明。

下面的规则大致说明了在什么时候应该使用什么搜寻方法，在稍后的小节中会继续讨论如何有效率地在结果数据集中搜寻数据。

1) 当使用Locate或是Lookup搜寻多个字段时，如果搜寻字段包含了索引字段，那么最好把索引字段放在最前面，这样可以加快搜寻的速度。

2) 同样，当使用TSimpleDataSet的Filter特性值来搜寻多个字段时，最好也把索引字段的条件值放在最前面，以提高搜寻的速度。

3) 如果想按照复杂的条件来搜寻结果数据集中的数据，那么可以使用TSimpleDataSet的OnFilterRecord事件处理函数。

4) 当结果数据集中的数据不多, 而且用户想按照特定的条件对数据进行分类时, 可以考虑使用过滤器来完成这个工作, 这样比较有效率。

上面的数个规则虽然可以合理地加快搜寻数据的速度, 但是当程序员需要处理大量的数据, 或是想要有更好的搜寻效率时, 就需要对于 dbExpress和DataSnap如何处理数据的原理有更多的认识, 以及使用更多的技巧。在下面的小节中将会继续讨论如何从dbExpress和DataSnap中压榨出更多的性能。

4.3 如何快速地在数据集中搜寻数据

前面的小节说明了如何使用 Delphi/Kylix提供的方法在结果数据集中搜寻数据, 程序员可以根据自己的需求使用不同的方法。但是使用这些方法搜寻数据并不等于程序员使用了快速方法搜寻数据。虽然 Locate和Lookup等方法在搜寻数据时会自动使用最快的方式, 但是当读者使用这些方法搜寻拥有大量数据的数据表时, 可能会惊讶地发现这将花上许多时间, 甚至缓慢到无法忍受的地步。这是为什么呢? 让我们使用下面的范例来说明。

下面的表格和图 4-22列出了当使用 Locate方法在不同数据量的数据表中搜寻数据时花费的时间。

被搜寻的数据表的记录量	搜寻时间 (秒)
数十个	0.102
500个	0.13
2000个	0.525
10 000个	2.803

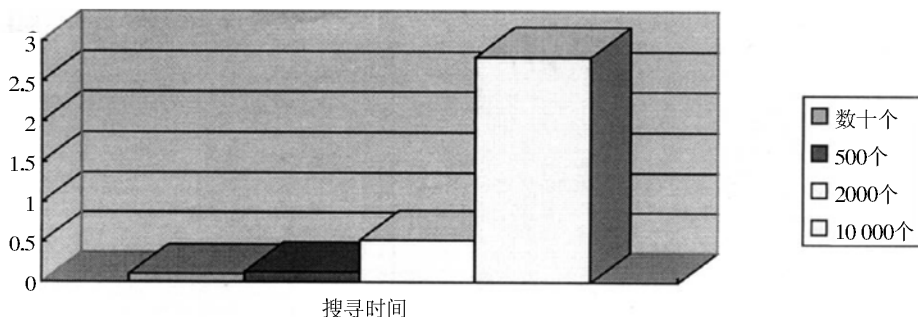


图4-22 在不同数据量的数据表中搜寻数据的时间

从上面的表格和图 4-22中我们可以很明显地看到, 数据表包含的数据越多, Delphi/Kylix搜寻数据的方法便越缓慢, 甚至当数据量超过一定的数量之后, 搜寻数据的时间更以数倍的幅度增加, 稍后会说明原因。

因此程序员必须知道，绝对不要在数据量较多的数据表中使用 **Locate**、**Lookup**或是过滤器的方法来搜寻数据，因为这样做不但速度缓慢，并且会造成网络以及客户端应用程序的极大负担。但是这似乎又陷入了进退两难的地步，如果不使用这些方法，那么如何在结果数据集中搜寻数据呢？

不用气馁，聪明的程序员总是有办法写出能够搜寻数据而且执行速度快的应用程序，下面的章节将逐一说明这些技巧。

4.3.1 分析Delphi/Kylix搜寻结果数据集方法的行为

为什么当使用**Locate**、**Lookup**等方法搜寻数据时，数据表包含的数据越多，搜寻的速度就越缓慢呢？当然，在理论上当数据更多时，搜寻的速度本来就应该比较缓慢，因为要搜寻的数据比较多。但是如果搜寻时间以数倍增长，那么就代表有一些事情不对劲了。

要知道**Locate**、**Lookup**如何在大量数据的数据表中搜寻数据，只需要开启**TSQLErrorMonitor**即可了解，下面的执行状态便是使用**Locate**在一个数据表中搜寻数据的情形：

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
Select * from PERFTEST order by ID
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
SELECT 0, ' ', ' ', A.RDB$RELATION_NAME, RDB$INDEX_NAME, RDB$FIELD_NAME,
B.RDB$FIELD_POSITION, 0, A.RDB$INDEX_TYPE, A.RDB$UNIQUE_FLAG, RDB$
CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE FROM RDB$INDICES, RDB$INDEX_SEGMENTS
FULL OUTER JOIN RDB$RELATION_CONSTRAINTS ON NAT.SRDB$RELATION_NAME =
C.RDB$RELATION_NAME AND RDB$CONSTRAINT_TYPE = 'PRIMARY' WHERE
(A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG AND (A.RDB$INDEX_NAME =
B.RDB$INDEX_NAME) AND (A.RDB$RELATION_NAME NOT IN ('PERFTEST'))) ORDER BY
RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
```

```
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
...
数千次的fetch
```

从上面的列表中我们可以分析出，当程序使用 **Locate**、**Lookup**等方法在结果数据集中搜寻数据时，**DataSnap**会先让**dbExpress**从数据源取得所有数据，再在结果数据集中搜寻数据，即使要搜寻的数据已经存在于结果数据集中，**DataSnap**仍然会先从后端数据源中将所有数据读取到客户端。

由于**dbxExpress**驱动程序版本不断在更新，因此读者使用 **TSQLMonitor**追踪的结果有可能与本书列出的有所不同。

这种执行行为模式虽然简单，但是对于拥有大量数据的数据表而言却非常没有效率。例如，如果数据表拥有 10万个记录，那么**DataSnap**会把10万个记录完全读到客户端，再进行搜寻。这样不但会花上许多时间，而且很有可能会使客户端的应用程序崩溃，因为这需要客户端使用大量内存。

因此要想有效率地在结果数据集中搜寻数据，我们就必须避免这种情形发生，让**DataSnap/dbExpress**不要把大量数据下载到客户端。在稍后的小节中会以一个实际的范例来说明如何一步一步地改善搜寻数据的效率。但是稍后讨论的技巧并不需要应用在所有的情况中，程序员应该根据数据表中数据的数量来决定。下面就分别讨论这些情形。

4 3 2 数据表包含少量的数据

对于包含数据较少的数据表，例如词组文件、职称文档等在数百个记录范围内的数据表，直接使用**Locate**、**Lookup**等方法搜寻数据并不会缓慢到什麼地步。而且这些数据通常不会被修改，程序员可以使用最经济的方式访问这些数据。因此建议直接使用**Locate**、**Lookup**等方法来搜寻数据，而无需使用稍后介绍的技术。

4 3 3 数据表包含大量的数据

如果一个数据表包含了大量的数据，例如数千、数万甚至是数 10个记录，那么程序员一定要避免直接使用**Locate**、**Lookup**等方法来搜寻数据。在这种情形中，程序员应该使用限制范围严密的SQL语句来先取得结果数据集，使之包含最少的数据量，

然后再使用 **Locate**、**Lookup** 等方法在结果数据集中搜寻数据。如果程序员使用宽松的 **SQL** 限制范围取得结果数据集，再使用 **Locate**、**Lookup** 等方法搜寻数据，那么便会下载大量的数据，同时造成网络的壅塞。

但是我们知道在许多应用中使用宽松的 **SQL** 语句来取得结果数据集是无法避免的，那么对于这种情形，程序员就需要使用各种技术来增加搜寻数据的效率。下面的小节就讨论了许多可以帮助程序员增加性能的技术。

4.3.4 快速搜寻数据

在本小节中将使用一个范例来观察并且改善搜寻数据的速度，并且也会观察这些不同的技术对于结果数据集中数据数量的影响。在下面的内容中，读者将会发现有一些技术可以大幅改善性能，不过天下没有免费的午餐，虽然这些技术可以加快速度，但是也会产生其他的副作用，程序员必须根据本身的需求结合不同的技术来达到最佳的效果。

首先让我们从最基础的地方开始，图 4-23 是本小节使用的范例应用程序主窗体。在这个主窗体中有 **Locate**、**Lookup**、**Filter** 等按钮，它们分别使用 **TClientDataSet** 的 **Locate**、**Lookup**、**Filter** 方法搜寻特定的数据，而 **Smart Locate**、**Smart Locate II** 则是稍后讨论的增加搜寻效率的按钮。右下角的 **TEdit** 则会显示在使用各种不同的搜寻方法时在结果数据集中产生的记录数量。而本范例搜寻的目标则是一个包含 1 万个记录的数据表。

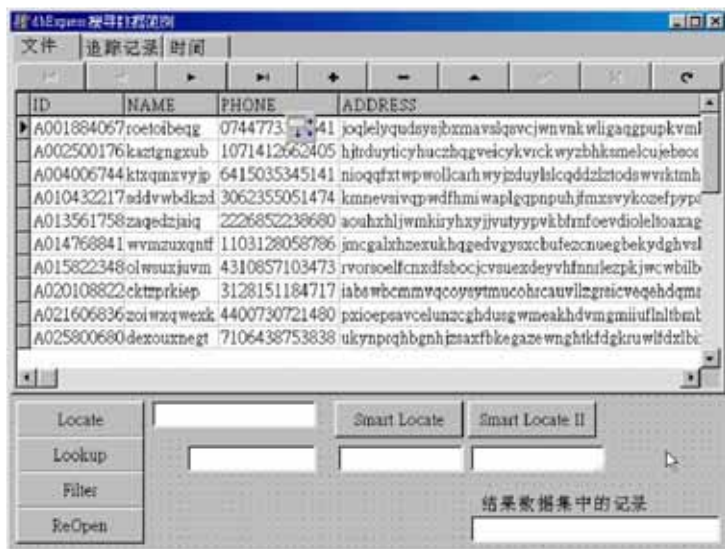


图4-23 搜寻数据范例的主窗体

首先让我们先使用 **Locate** 来搜寻数据。在下面的范例中连接了一个拥有大量数据

的InterBase数据库。本范例设置应用程序中 TClientDataSet的PacketRecords特性值为10，即一次访问10个数据，而我们要搜寻的数据是否已经存在于 TClientDataSet中的数据，根据这个范例我们可清楚地了解 dbExpress搜寻数据的行为。

1. 使用TClientDataSet的Locate、Lookup和Filter方法搜寻数据

范例应用程序主窗体中的 Locate按钮使用下列的程序代码搜寻范例数据表中的键值字段ID，现在读者应该很熟悉 Locate的语法，因此就不再多做说明了。

```
procedure TfrmFindData.btnLocateClick(Sender: TObject;
begin
    GetStartTime;
    dmFindData.sqlcdsTest.Locate('ID', edtID.Text, loCaseInsensitive,
    loPartialKey);
    GetEndTime;
    AppMsg('Locate', GetRunTime);
    edtTime1.Text := FloatToStr(GetRunTime) / 1000.0;
    edtRecordCount.Text := IntToStr(dmFindData.sqlcdsTest.RecordCount);
end;
```

现在执行应用程序，在主窗体中 Locate按钮旁的 TEdit控件中输入已经存在于 TClientDataSet中的A025800680记录。由于这个记录已经存在于结果数据集中（请参考图4 23，它是TClientDataSet中的第10个记录），因此我们会认为 TClientDataSet应该可以瞬间找到这个记录，并且不需要再从后端数据源中取得额外的数据。

不过图4 24是点击 Locate按钮执行数据搜寻的结果，从图4 24中我们发现 TClientDataSet并不是这样搜寻数据的，而是先从后端数据源中取得所有数据，再进行搜寻工作，因此居然需要花上2.803秒的时间搜寻就在我们眼前的数据。说实话，这种执行行为实在非常愚蠢，不过这是 Locate、Lookup、Filter等方法的默认行为，但是聪明的程序员却可以改善它。

在我们继续讨论之前，先看看 Locate、Lookup和Filter等方法在性能上有没有什么差别。下表是使用不同的方法搜寻相同数据的结果，从表中我们可以知道它们的搜寻效率几乎没有什么明显的差别。

	Locate	Lookup	Filter
搜寻时间	2.803	2.856	2.901

不过，当所有数据被下载到 TClientDataSet的结果数据集中后，搜寻数据的速度便明显加快了许多。例如，如果我们再次搜寻 A025800680记录，那么Locate便只需要花0.01秒的时间，相当快。接下来的目标便是改善 Locate初次搜寻数据的效率。

改善Locate等搜寻数据的效率的第一个方法是设法避免 TClientDataSet总是要把后端数据源中所有数据一次下载到客户端的执行行为。让 TClientDataSet先在已经存在于结果数据集中的数据中进行搜寻，例如 A025800680记录已经存在于结果数据集

中，因此如果可以让 TClientDataSet 先直接在结果数据集中进行搜寻，那么就可以大幅增加效率。



图4-24 使用Locate方法搜寻数据的结果

但是要如何切断 TClientDataSet 与后端数据源的连接呢？TClientDataSet 已经提供了一个方法，那就是 CloneCursor。

2. 使用CloneCursor搜寻数据

TClientDataSet/TSimpleDataSet 的 CloneCursor 方法可以让另外一个 TClientDataSet/TSimpleDataSet 分享相同的数据。但是 CloneCursor 另外一个没有说明的功能则是可以避免 TClientDataSet/TSimpleDataSet 从后端数据源下载所有数据。通过这个隐藏的功能，我们可以先使用 CloneCursor 方法在结果数据集中搜寻数据，如果没有发现需要的数据，那么再让原先的 TClientDataSet/TSimpleDataSet 使用 Locate、Lookup 等方法继续搜寻数据。这对于搜寻已经存在于结果数据集中的数据而言是非常有效率的。

因此范例应用程序的 Smart Locate 按钮便使用了下面的程序代码来搜寻数据：

```
procedure TfrmFindData.btnSmartLocateClick(Sender: TObject);
var
  aCDS : TClientDataSet;
begin
  aCDS := TClientDataSet.Create(Self);
  try
    GetStartTime;
    aCDS.CloneCursor(dmFindData.sqlcdsTest, True);

    if aCDS.Locate('ID', edtID.Text, [loCaseInsensitive, loPartial]) then
      dmFindData.sqlcdsTest.MoveBy(aCDS.RecNo - dmFindData.sqlcdsTest.RecNo
```

```

else
    dmFindData.sqlcdsTest.Locate('ID',edtID.Text,loCaseInsensitive,
    loPartialKey]);

GetEndTime;
AppMsg('Locate', GetRunTime);
edtTime2.Text := FloatToStr(GetRunTime) / 1000.0;
edtRecordCount.Text := IntToStr(dmFindData.sqlcdsTest.RecordCount);
finally
    aCDS.Free;
end;
end;
end;

```

上面的程序代码首先建立一个 TClientDataSet 组件 aCDS，再调用 CloneCursor 方法从原先的 TSimpleDataSet 组件 sqlcdsTest 取得另外一个游标，然后调用 aCDS 的 Locate 在结果数据集中搜寻数据。由于 aCDS 会避免 sqlcdsTest 从后端数据源下载所有数据，因此它的搜寻速度非常好。图 4-25 便是使用 CloneCursor 搜寻 A025800680 记录的结果，我们可以看到它只需要 0.004 秒，比直接使用 sqlcdsTest 的 Locate 快了数百倍。

当然，如果 aCDS 无法在结果数据集中找到需要的数据，它便会再把搜寻工作交还给 sqlcdsTest，使用一般的 Locate 方法来搜寻数据。

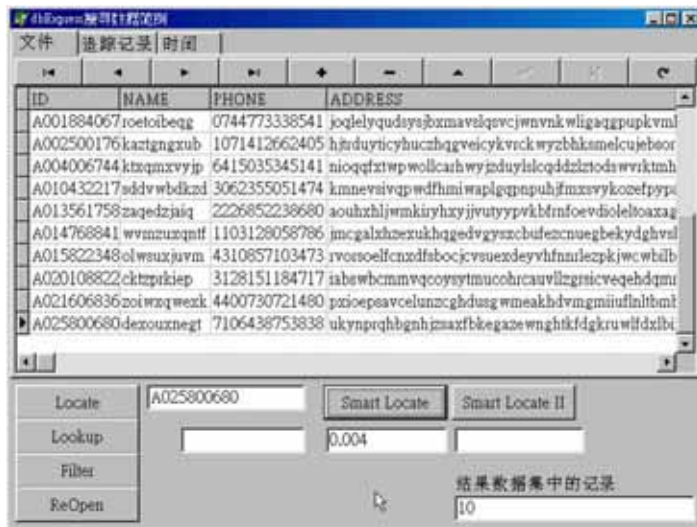


图4-25 使用CloneCursor方法搜寻数据的结果

使用 CloneCursor 方法可以让程序员巧妙地避免下载所有数据的负荷，但是当要搜寻的数据并不存在于结果数据集中时，CloneCursor 也无法避免下载所有数据。当这种情形发生时，使用 CloneCursor 方法反而比直接使用 TClientDataSet/TSimpleDataSet 的 Locate 方法慢。

让我们继续使用其他技巧来设法解决这个问题。

3. 使用SQL语句搜寻数据

我们从上一小节中知道 **CloneCursor** 也无法避免 **TClientDataSet/TSimpleDataSet** 下载所有数据的情形，因此我们终究是要避免这种情形发生的可能，才能够保证在大部分的时候（也许是 99.99% 的情形中）可以大幅增加搜寻数据的效率。

先想想 **CloneCursor** 方法在什么时候仍然需要下载所有数据？那就是当欲搜寻的数据不存在于结果数据集中时。如果我们能够在这种情形发生时，直接从数据源中取得要搜寻的单个记录，然后把它加入到 **TClientDataSet/TSimpleDataSet** 的 **Data** 特性中，不就可以解决这个问题了吗？这的确是一个非常好的点子，因为同时使用 **CloneCursor** 和这种技巧，的确可以在 99.99% 以上的情形中大幅增加性能。现在看看如何实现这种功能。

范例应用程序主窗体中的 **Smart Locate II** 按钮使用了如下的程序代码来搜寻数据：

```
procedure TfrmFindData.btnSmartLocate2Click(Sender: TObject);
var
    aCDS : TClientDataSet;
begin
    aCDS := TClientDataSet.Create(Self);
    try
        GetStartTime;
        aCDS.CloneCursor(dmFindData.sqlcdsTest, True);
        if aCDS.Locate('ID', edtID.Text, [loCaseInsensitive, loPartialKey]) then
            dmFindData.sqlcdsTest.MoveBy(aCDS.RecNo - dmFindData.sqlcdsTest.RecNo, 1)
        else
            begin
                dmFindData.sqlcdsTemp.Active := False;
                try
                    dmFindData.sqlcdsTemp.Params.ParamByName('ID1').Value := edtID.Text;
                    dmFindData.sqlcdsTemp.Active := True;

                    if (dmFindData.sqlcdsTemp.RecordCount) > 0 then
                        begin
                            dmFindData.sqlcdsTest.AppendData(dmFindData.sqlcdsTemp.Data, False);
                            dmFindData.sqlcdsTest.MoveBy(dmFindData.sqlcdsTest.RecordCount, 1);
                            dmFindData.sqlcdsTest.RecNo;
                        end;
                    finally
                        dmFindData.sqlcdsTemp.Active := False;
                    end;
                end;
            end;
    end;
```

```

GetEndTime;
AppMsg ('Locate', GetRunTime
edtTime3.Text := FloatToStr(GetRunTime) / 1000.0 ;
edtRecordCount.Text := IntToStr(FindData.sqlcdsTest.RecordCount);
finally
aCDS.Free;
end;
end;

```

上面的程序代码首先使用前面介绍的 **CloneCursor** 方法在结果数据集中搜寻数据，但是当 **CloneCursor** 无法在结果数据集中找到欲搜寻的数据时，它就使用另外一个 **TSimpleDataSet** 组件，直接通过 SQL 语句从后端数据源搜寻这个记录。如果找到的话，就调用原先 **TSimpleDataSet** 的 **AppendData** 方法把找到的记录加入其中，并且把游标移动到这个记录上。

图 4 26 便是使用这个技巧搜寻数据的结果，从图 4 26 中可以看到由于搜寻的是 A025800680 记录，因此它是使用 **CloneCursor** 技巧来找到这个记录的，速度是非常快的。稍后我们会搜寻不在目前结果数据集中的其他数据，看看结果会如何。

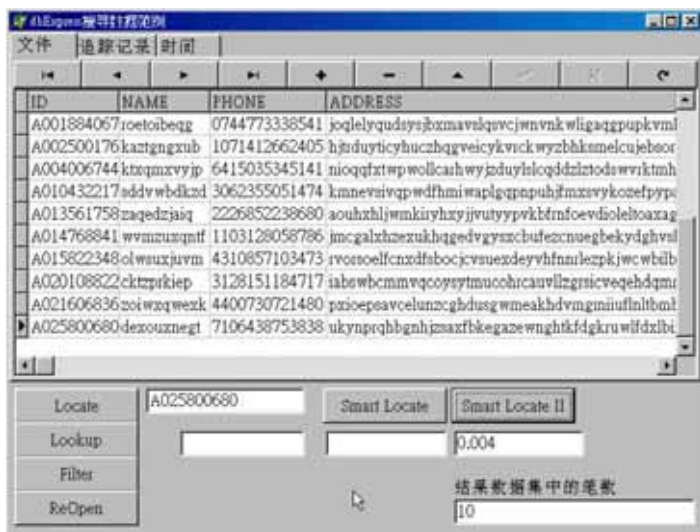


图4-26 使用CloneCursor方法和SQL语句搜寻数据的结果

在我们继续讨论之前，先看看只是简单地使用 **CloneCursor** 方法就可以将性能提高数百倍，如图 4 27 所示。

现在让我们继续搜寻一些目前不存在于结果数据集中的数据，看看不同的搜寻方式是否有任何差异。

4. 搜寻后部数据

现在让我们搜寻位于数据表中后部的数据。由于这个记录现在不在结果数据集中，

因此 TSimpleDataSet 必须从后端数据源中取得数据。那么直接使用 Locate、使用 CloneCursor 或是使用 CloneCursor 加 SQL 语句是否会有什么差别呢？

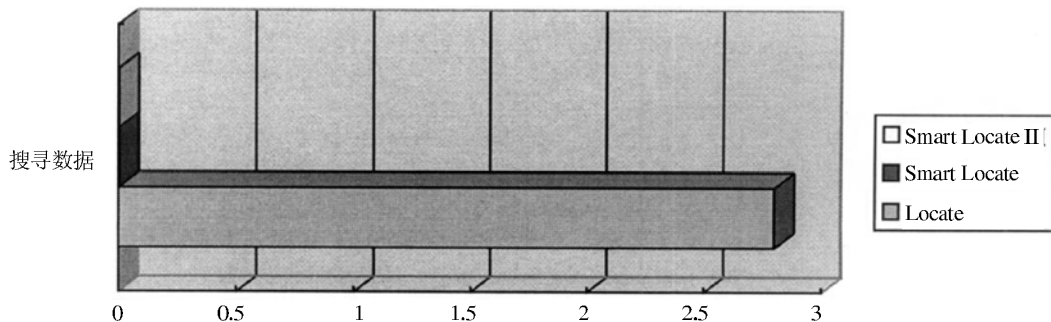


图4-27 三种数据搜寻方法的性能比较

图4 28是再次执行范例应用程序并且搜寻 W145812312记录的结果，直接使用 Locate 的执行结果与搜寻已经存在于结果数据集中的数据几乎没有什么不一样，因此在直接使用 Locate、Lookup 和 Filter 等方法搜寻数据时，不管要搜寻的数据是已经存在于结果数据集中，或是还在后端数据源中，所花的时间几乎都一样，非常稳定。

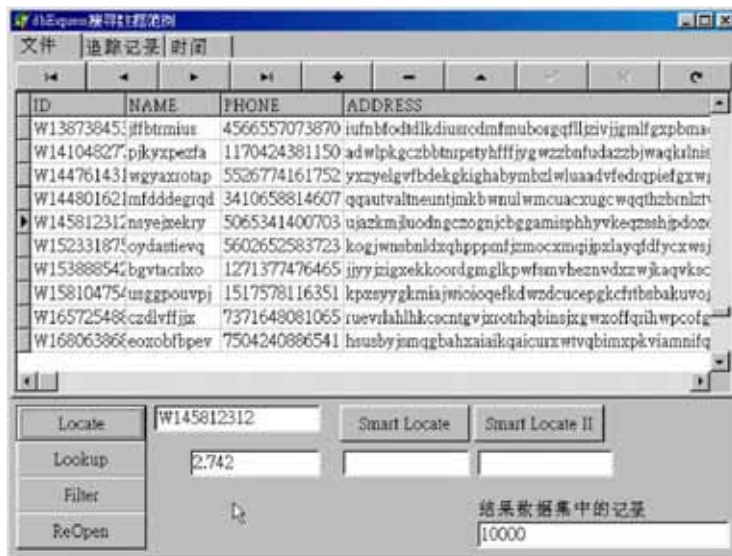


图4-28 使用Locate方法搜寻后部数据的结果

如果使用 CloneCursor 方法搜寻不存在于结果数据集中的数据，那么结果如图 4 29所示，比直接使用 Locate 等方法差了许多。性能下降是因为由于现在这个记录不在结果数据集中，因此 CloneCursor 方法仍然会再使用 Locate 方法搜寻数据，因此至少需要花上与直接使用 Locate 方法一样的时间。不过由于此时 TSimpleDataSet 需要维护两个游标，因此速度会更缓慢，这就是需要花上 5秒多时间的原因。

请注意，这时 CloneCursor 方法也把后端数据源中的所有数据下载到客户端，并没有少下载任何记录。

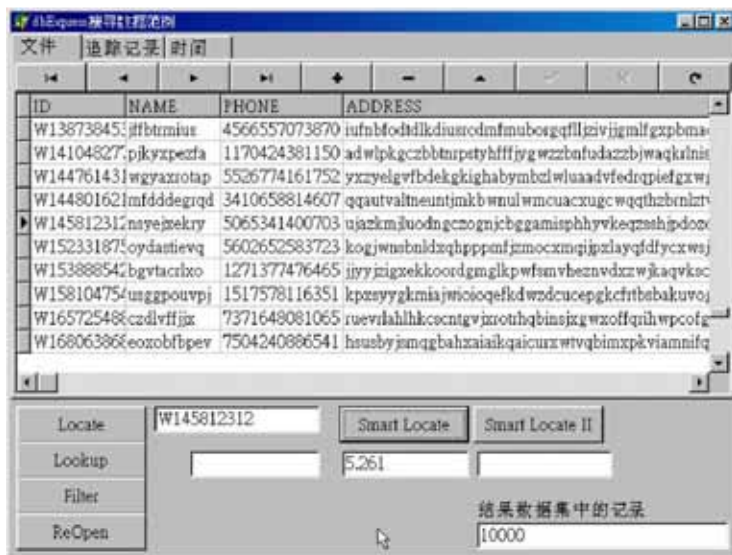


图4-29 使用CloneCursor方法搜寻后部数据的结果

最后，让我们使用 Smart Locate II 的方法来搜寻数据，由于此时这个记录不在结果数据集中，因此 Smart Locate II 会使用 SQL 语句直接在后端数据源中搜寻数据，再把搜寻到的数据加入到客户端的 TSimpleDataSet 中。图 4 30 是搜寻 W145812312 记录的结果。

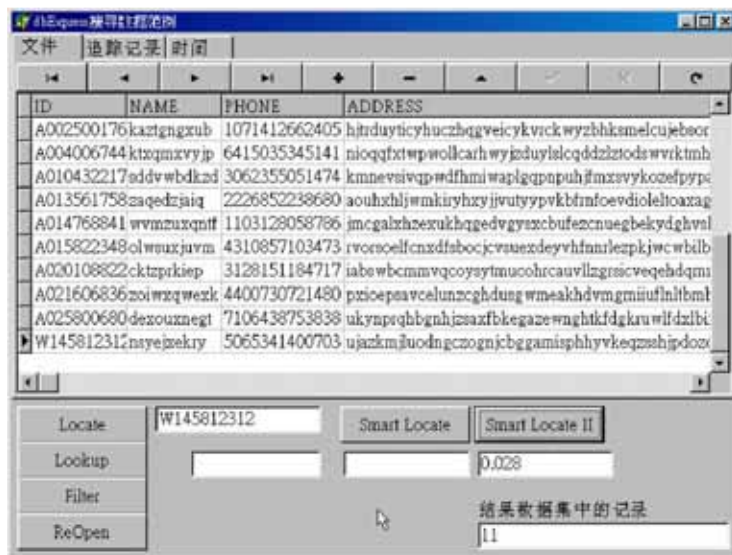


图4-30 使用CloneCursor加SQL语句的方法搜寻后部数据的结果

请注意,使用这种方法搜寻数据时性能仍然是最好的,比刚才的两种方法快了数百倍。同时这种方法会让结果数据集中维持最少的数据。例如,前面的两种方法都会下载所有数据,但是这个方法却只会下载一个记录,因此此时结果数据集中只有11个记录,而前面的两种方法却下载了10 000个记录,之间的差异是很大的。如果在一个网络环境中,客户端有许多用户同时执行系统,那么这种方法可以大幅降低网络的负荷。

图4 31是使用三种方法搜寻不在结果数据集中的数据的结果,请和前面的图 4 27比较一下,我们会发现直接使用 **Locate**等方法需要花的时间是一样的,非常稳定。**CloneCursor**方法则大幅降低了性能。至于 **CloneCursor**加SQL语句仍然是最快速的,而且会下载最少的数据量。

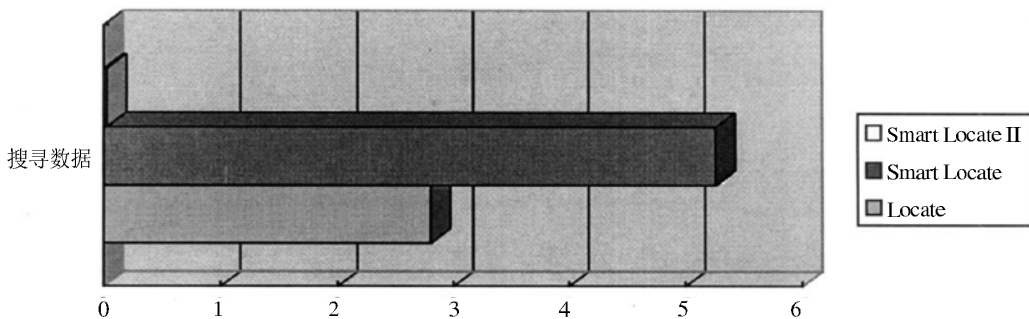


图4-31 三种方法搜寻后部数据的结果

使用**CloneCursor**加SQL语句的方法虽然是最快速的,但是这种方法也有副作用。那就是如果以后用户一直访问随后的数据,那么当 **TClientDataSet/TSimpleDataSet**访问到包含刚才直接使用SQL语句获得的记录的**PacketRecords**时,那么**TClientDataSet/TSimpleDataSet**组件中便会包含重复的数据。不过这个情形在数据量大的应用中并不会经常发生,因此这种技术仍然值得程序员使用。即使会发生这种情形,我们仍然可以用非常有效率的方法来克服它,稍后会继续说明。

5. 使用TClientDataSet的Data特性来搜寻数据

到这里为止,范例应用程序的 **Smart Locate**方法可能会让程序员陷入两难的境地中,因为**CloneCursor**方法虽然可以快速地在结果数据集中找到用户欲搜寻的数据,但是当要搜寻不在结果数据集中的数据时,却需要花上更多的时间。那么到底有没有更好的方法可以让搜寻结果数据集中的数据非常快速,搜寻不在结果数据集中的数据也能具有与直接使用 **Locate**等方法类似的效率,而且也没有使用 **CloneCursor**加SQL语句的副作用呢?

还是有的,那就是直接使用 **TClientDataSet/TSimpleDataSet**组件的**Data**特性值。当程序要搜寻数据时,我们可以先把原先 **TClientDataSet/TSimpleDataSet**组件的**Data**特性值拷贝给另外一个新的**TClientDataSet/TSimpleDataSet**组件的**Data**特性值,再在

新的TClientDataSet/TSimpleDataSet组件中使用Locate等方法搜寻数据，这样不但可以切断与后端数据源的连接，还可避免下载所有数据，之后再在新的 TClientDataSet/TSimpleDataSet组件中搜寻数据。这样可以快速地在结果数据集中搜寻数据，如果没有找到的话，那么就再要求原先的 TClientDataSet/TSimpleDataSet组件在后端数据源中搜寻数据。因此我们再加入另外一个 Smart Locate I按钮，并且使用如下的程序代码来搜寻数据：

```

procedure TfrmFindData.btnSmartLocate1Click (Sender: TObjecT;
var
    aCDS : TClientDataSet;
begin
    aCDS := TClientDataSet.CreateSelf);
    GetStartTime;
    // aCDS.CloneCursOrd dmFindData.sqlcdsTest, True
    aCDS.Data := dmFindData.sqlcdsTest.Data;

    if aCDS.Locate ('ID', edtID.Text, [loCaseInsensitive, loPartialKey] then
        dmFindData.sqlcdsTest.MoveBy (aCDS.RecNo - dmFindData.sqlcdsTest.RecNo
    else
        begin
            aCDS.Free;
            dmFindData.sqlcdsTest.Locate ('ID', edtID.Text, [loCaseInsensitive,
                loPartialKey]);
        end;

        GetEndTime;
        AppMsg ('Locate', GetRunTime);
        edtTime2.Text := FloatToStr(GetRunTime) / 1000.0;
        edtRecordCount.Text := IntToStr(dmFindData.sqlcdsTest.RecordCount);
    end;

```

上面的程序代码把原始数据集中的数据（Data特性）指定给另外一个暂时的TClientDataSet（aCDS），再对aCDS进行搜寻数据的工作。如果在aCDS中找到了数据，便让数据集sqlcdsTest目前的记录指针移动到此记录上。如果在aCDS中没有找到用户欲搜寻的数据，这代表欲搜寻的数据尚未下载到客户端，那么就直接让数据集sqlcdsTest的Locate来进行搜寻。

现在让我们使用另外一个拥有更多数据的数据表进行测试，这个测试数据表包含50000个记录，以便了解将数据拷贝到另外一个 TClientDataSet/TSimpleDataSet是否需要花上大量时间，只有在这个动作只需要少量时间的情况下它才值得我们使用。

图4 32到图4 35是分别搜寻已经存在于结果数据集中的数据，以及搜寻还不存在于结果数据集中的数据的结果。

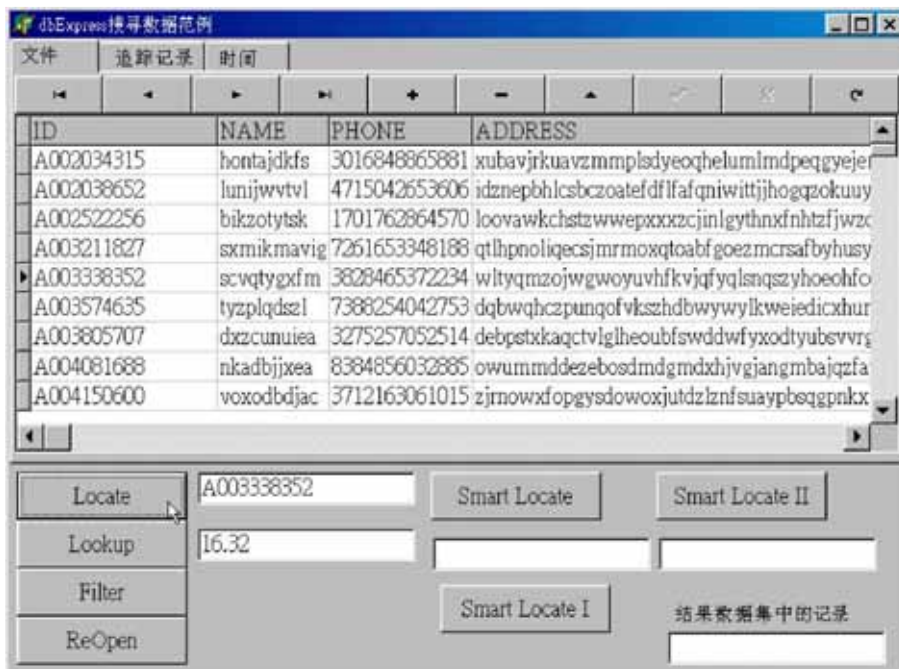


图4-32 直接使用Locate搜寻数据

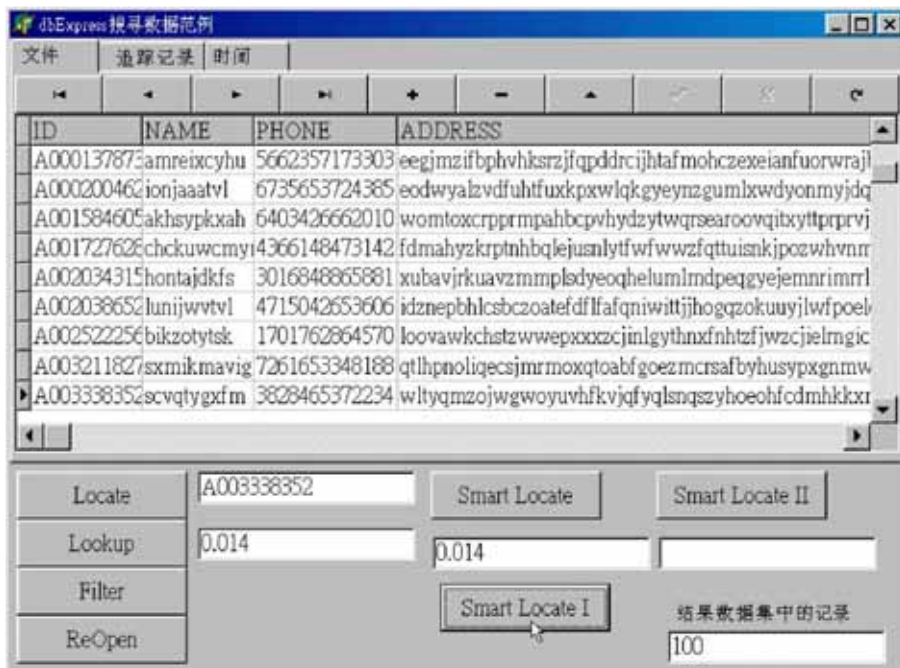


图4-33 通过TClientDataSet/TSimpleDataSet的Data特性值搜寻数据

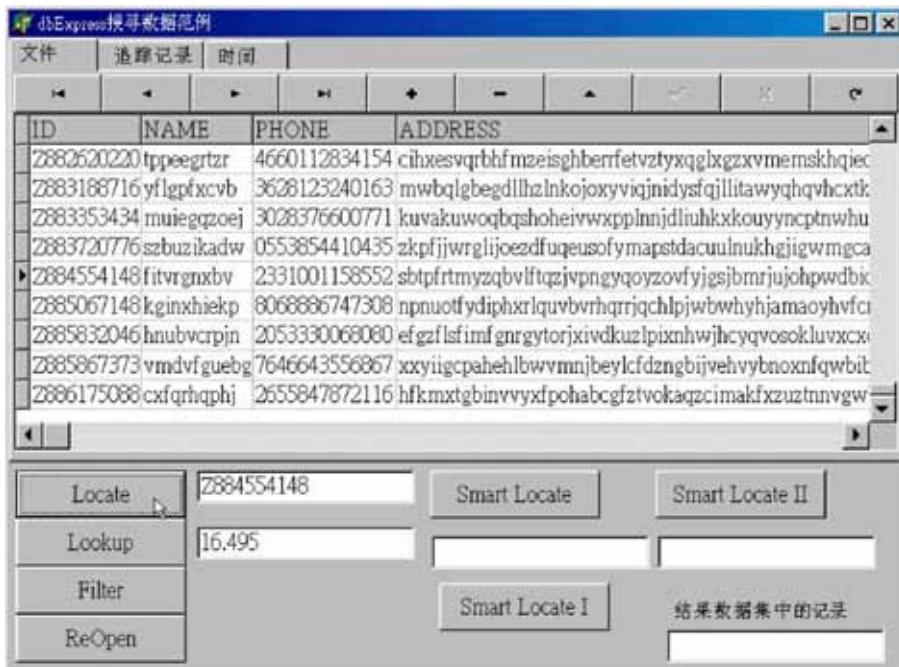


图4-34 直接使用Locate搜寻不在结果数据集中的数据

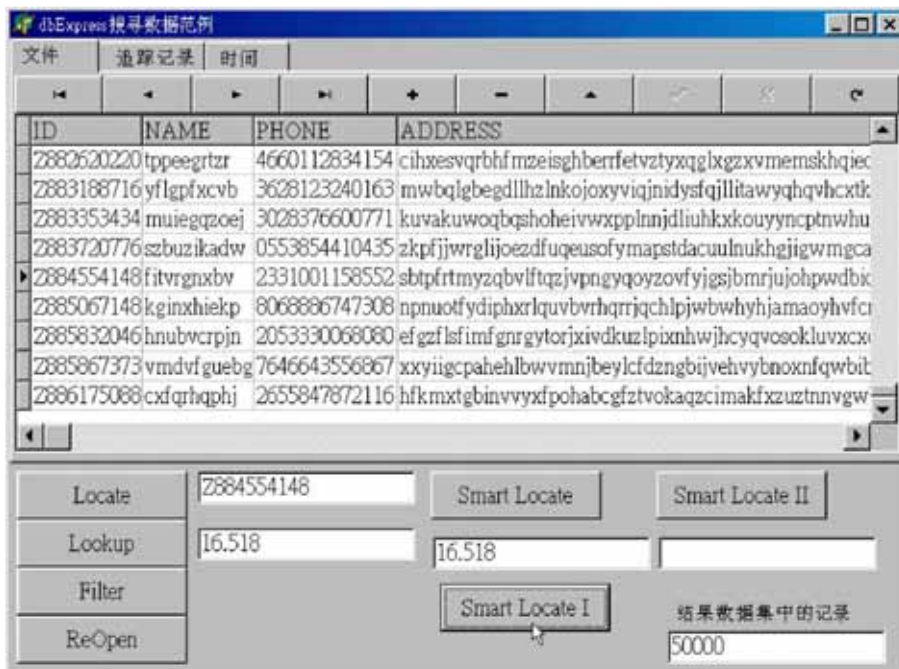


图4-35 通过TClientDataSet/TSimpleDataSet的Data特性搜寻不在结果数据集中的数据

图4 36是通过Data特性值搜寻数据的比较图,从图4 36中可以看到使用Data特性值来搜寻数据的确可以获得良好的性能。当搜寻已经存在于结果数据集中的数据时,它可以提供比直接使用Locate方法好数百倍的效率。而在搜寻不存在于结果数据集中的数据时,它可以提供与直接使用Locate方法一样的性能,而没有明显的差异。这个技巧融合了CloneCursor的优点,而又没有CloneCursor+SQL的缺点。

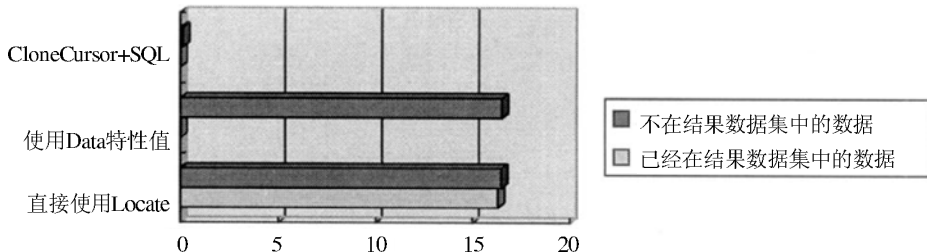


图4-36 三种搜寻数据方法的结果比较

从这个测试中我们可以推知一个重要的DataSnap/dbExpress执行行为,那就是当我们拷贝一个TClientDataSet/TSimpleDataSet的Data特性值时,DataSnap/dbExpress似乎只是像COM/DCOM那样增加了原先Data特性值的引用计数值,而没有真正地在内存中拷贝大量数据,因此这个搜寻技巧才能够在搜寻不存在于结果数据集中的数据时,提供与直接使用Locate方法一样的效率。也是因为这种特性,这个技巧才值得我们使用。

下面的表格列出了四种数据搜寻方法的优缺点,程序员可以根据应用程序的需要选择适当的搜寻方法。

搜寻方式	缺 点	优 点
Locate	会从数据源中下载所有数据,如果数据表中包含大量数据,那么性能会很差,而且容易造成客户端死机	搜寻数据的性能保持稳定的结果,而不管要搜寻的数据是否已经存在于结果数据集中,而且没有其他副作用
使用CloneCursor	如果欲搜寻的数据不在结果数据集中,那么仍然会下载数据表中的所有数据,而且表现得比直接使用Locate等方法还缓慢	如果欲搜寻的数据已经存在于结果数据集中,那么它可以避免下载所有数据,并且能够快速地找到要搜寻的数据
使用CloneCursor 加SQL语句加AppendData	可能会发生数据重复的副作用	提供最高的搜寻效率,不管数据是否已经在结果数据集中。同时,下载的数据量最少,网络的负荷最轻
使用拷贝数据	程序必须拷贝一些额外的数据	可以提供比直接使用Locate等方法更好或相同的搜寻效率,同时没有CloneCursor方法的缺点,也没有CloneCursor加SQL语句的副作用

6. 避免CloneCursor加SQL语句产生数据重复的情形

在前面讨论 CloneCursor+SQL 的搜寻技巧时我们了解到, 这种方式虽然是最迅速的, 但是却有可能产生数据重复的情形, 这似乎有一点儿美中不足。事实上要解决这个问题也不困难, 只要我们再结合一些小技巧就可以克服了。

首先先使用一个 TClientDataSet 拷贝原先 TClientDataSet/TSimpleDataSet 的 Data 特性值, 接着当原先的 TClientDataSet/TSimpleDataSet 访问每一个随后的 PacketRecords 时, 先在拷贝的 TClientDataSet 中使用 Locate 一一检查 PacketRecords 中的数据是否已经存在了。如果是的话, 那么就先删除 PacketRecords 中的这个记录。最后当 PacketRecords 中的所有数据都检查完毕之后, 再使用 AppendData 把数据加入到原先的 TClientDataSet/TSimpleDataSet 中, 这样就不会产生数据重复的问题了。如果你没有把 PacketRecords 特性值设得太大, 例如只设为 10, 那么这种数据重复检查的负荷将会非常小, 因此不会影响性能。

从本小节的讨论中可以知道, 程序员在了解了 dbExpress 的工作原理之后, 再配合一些程序技巧就可以让搜寻数据的速度大幅提升。不过本小节讨论的增加搜寻效率的技巧仍然需要程序员根据自己数据库应用系统的需要选择使用, 没有一种技巧能够在各种数据情形中都大幅增加性能。除了搜寻数据的性能之外, 在稍后的章节中将会进一步讨论如何有效率地处理修改的数据。

4.4 结论

本章讨论的内容是非常重要的主题, 因为对于许多数据库应用系统来说搜寻数据几乎是最常执行的工作。一般来说, 数据库应用程序可以使用两种方式来搜寻用户需要的数据, 第一种是不管搜寻什么数据都直接使用 SQL 语句向数据库要求返回符合条件的数据。第二种方式则是在已经由数据库服务器返回的结果数据集中搜寻数据。Delphi/Kylix 提供了各种机制允许程序员使用上述的两种方式来搜寻数据, 本章讨论的重点放在如何在结果数据集中搜寻数据。

虽然 Delphi/Kylix 的 dbExpress 组件提供了数个不同的方法让程序员在结果数据集中搜寻数据, 但是这些方法各有不同的使用时机, 程序员应该使用最适当的方法来搜寻数据。由于 dbExpress 在结果数据集中搜寻数据的方式是通过先把所有数据下载到客户端的结果数据集中, 再在结果数据集中搜寻数据, 因此当数据表拥有大量数据时这种搜寻数据的方式便非常没有效率, 同时把大量数据取到客户端也会造成严重的网络负荷, 甚至会造成客户端应用程序死机。因此程序员必须了解 dbExpress 搜寻数据的执行行为而且使用聪明的方法来搜寻数据。

本章除了详细说明 dbExpress 搜寻数据的执行行为之外, 也讨论了数个有效率的

数据搜寻方法，使用这些方法可以让应用程序搜寻数据的时间大为降低，并且有效地减少从后端数据表下载到客户端的数据量。

本章说明了如何有效率地在结果数据集中搜寻数据，在稍后的章节中也将说明如何使用 **dbExpress** 有效率地处理数据，以及如何有效率地使用 **SQL** 语句来搜寻数据。在了解了这些重要的知识之后，程序员将可以使用 **dbExpress** 编写出非常有效率的数据库应用系统。

第5章 dbExpress的高级技术

阅读完了前面的章节之后，现在读者应该了解并且掌握了许多 dbExpress技术。对于编写一般的 dbExpress数据库应用程序来说，应该是没问题了。但是前面章节着重的一些 dbExpress使用技巧和高级概念。对于数据库应用系统来说，除了需要知道这些 dbExpress程序技术之外，程序员也必须要了解许多应用程序在处理数据时使用的重要概念，例如数据库事务管理（Transaction Management），如何处理修改数据时发生的错误等。因为这些对数据处理很重要的概念除了会影响数据的完整性和正确性之外，对于 dbExpress应用程序的性能也会有相当的影响，因此这些数据处理的观念和技巧也是非常重要的。

本章讨论的内容就在于介绍这些重要的数据处理概念，让读者除了有良好的程序技巧之外，也能够掌握这些重要的数据处理概念，如此一来在开发数据库应用系统时才能够更得心应手。此外，本章也将介绍如何在 COM+中使用 dbExpress来访问数据，让 dbExpress搭配 COM+ 组件开发 Microsoft DNA 类型的应用。

5.1 事务管理

事务管理（Transaction Management）是目前所有数据库软件都必须支持的重要功能之一。原本事务管理只出现在较大型的关系型数据库中，但是现在事务管理功能也被大多数基于文件的数据库支持。例如，原本简单的以有效率为主要目标的 MySQL 现在也支持事务管理。这是为什么？因为事务管理对于数据一致性的保证是任何关键性数据库应用系统都必须提供的功能。

数据库提供的事务管理必须提供四个最基本的功能，那就是

- **Atomic（原子性）**：事务管理必须完全执行，或是完全不被执行。
- **Consistent（一致性）**：事务管理会维护数据库的内部一致性。
- **Isolated（隔离性）**：事务管理可保证数据库中似乎只有目前的事务在执行，而没有其他事务。
- **Durable（持久性）**：事务管理的执行结果即使是在数据库发生错误时也不会遗失。

这四个功能统称为 ACID（取以上每个功能的第一个大写字母）法则。在目前的关系型数据库中，ACID 已经被广泛支持，而且关系型数据库也提供 API 让客户端调用以便自动享受事务管理提供的功能，以保证数据的一致性和完整性。

dbExpress 的 TSQLConnection 组件封装了每一个数据源提供的事务管理功能，并

且通过提供数个方法和特性值让程序员能够通过 **dbExpress**来控制数据源的事务管理功能。程序员可以调用 **TSQLConnection**组件的**StartTransaction**激活数据源的事务状态, 然后进行数据库的修改工作, 最后调用 **TSQLConnection**组件的**Commit**方法以确保在事务中修改的数据被正确地更新回数据源, 或是在发生错误时调用 **TSQLConnection**组件的**Rollback**以取消事务。这些方法的原型如下:

```
procedure StartTransaction (TransDesc: TTransactionDesc);
procedure      Commit (TransDesc: TTransactionDesc);
procedure      Rollback (TransDesc: TTransactionDesc)
```

TSQLConnection组件的这三个方法都接受一个 **TTransactionDesc**类型的参数, 而 **TTransactionDesc**具有如下的定义:

```
TTransIsolationLevel(xilDIRTYREAD, xilREADCOMMITTED, xilREPEATABLE,
xilCUSTOM);
TTransactionDesc packed record
    TransactionID      : LongWord;
    GlobalID           : LongWord;
    IsolationLevel     : TTransIsolationLevel;
    CustomIsolation    : LongWord;
end;
```

由于在**dbExpress**支持的数据源中有的数据源支持嵌套事务, 有的数据源支持多重事务, 因此当程序员调用 **StartTransaction**激活数据源的事务时, 必须通过 **TTransactionDesc**类型的参数来指定事务的特性, 例如激活的事务 ID, 以便稍后调用 **Commit**或是**Rollback**来确保特定的事务成功执行或是取消。 **TTransactionDesc**中的 **IsolationLevel**字段则是让程序员指定事务使用的数据隔离程度。下面的表格说明了 **TTransactionDesc**中每一个字段的意义:

TTransactionDesc	意 义
TransactionID	由程序员指定的唯一 ID, 代表目前的事务。这个程序员指定的事务 ID不能与目前在数据库连接中执行的任何事务 ID相同
GlobalID	只在Oracle中使用的全局事务ID
IsolationLevel	这个字段代表不同事务之间的数据隔离程度, 在下一小节中会详细说明
CustomIsolation	如果IsolationLevel字段的值被指定为 xilCUSTOM, 那么程序员必须通过 CustomIsolation字段指定使用哪种定制的IsolationLevel

TTransactionDesc中的**IsolationLevel**字段的意义会在下一小节中详细说明。

因此, 在一般的 **dbExpress**应用程序中如果程序员需要激活事务, 以便开始执行需要保证数据一致性和完整性的修改工作, 那么可以使用如下的程序代码:

```
var
    aTD : TTransactionDesc;
begin
```

```

aTD.TransactionID := 1;
aTD.IsolationLevel := xilREADCOMMITTED;
Self.SQLConnection1.StartTransaction (aTD);
try
    /执行数据库修改工作
except
    Self.SQLConnection1.Rollback (aTD);
end;
Self.SQLConnection1.Commit (aTD);
...
end;

```

上面的程序代码首先声明一个 **TTransactionDesc** 类型的变量 **aTD**，接着在程序块中先指定要激活的事务 ID，再指定使用的 **IsolationLevel**，最后调用 **TSQLConnection** 的 **StartTransaction** 方法以激活数据源的事务功能。在调用 **StartTransaction** 方法之后的数据修改都保证会成功地更新回数据源，或是在发生错误时将数据库状态恢复到激活事务之前的状态。如果在整个修改数据的过程中没有发生任何错误，那么程序员可以调用 **Commit** 以确保事务更新修改的数据；如果发生任何错误，就在 **except** 语句中调用 **Rollback** 取消事务修改的数据。

除了刚才介绍的 **dbExpress** 事务方法之外，**TSQLConnection** 组件也提供了数个特性让程序员可以检查数据源提供的事务特性。例如 **TSQLConnection** 的 **InTransaction** 特性值让程序员可以检查目前是否已经激活了事务，**TransactionsSupported** 特性值让程序员可以检查目前 **dbExpress** 连接的数据源是否支持事务。另外 **MultipleTransactionsSupported** 特性值让程序员可以检查目前 **dbExpress** 连接的数据源是否支持嵌套事务。这些特性的声明原型如下：

```

property InTransaction: Boolean read GetInTransaction;
property TransactionsSupported: LongBool read FTransactionsSupported;
property MultipleTransactionsSupported: LongBool read FSupportsMultiTrans;

```

所谓嵌套事务是指程序员可以在激活一个事务之后，在这个事务中再激活另外一个事务。由于 **TTransactionDesc** 的 **TransactionID** 字段让程序员可以指定不同的 ID 值，因此程序员可以将不同的值指定给 **TransactionID** 字段，以此识别不同的事务。不过程序员必须知道 **dbExpress** 支持的所有数据源并不都支持嵌套事务，因此程序员可以通过 **MultipleTransactionsSupported** 特性值来检查目前的数据源是否支持嵌套事务。

下面的程序代码使用了刚才介绍的特性来判断目前的数据源是否支持事务，以及目前是否已经激活了事务。如果数据源支持事务，而且目前尚未激活事务，那么才调用 **StartTransaction** 以激活事务：

```

if (Self.SQLConnection1.TransactionsSupported) then
begin

```



```
if (not Self.SQLConnection1.InTransaction) then
    Self.SQLConnection1.StartTransaction (aTD);
...
//事务程序代码
end;
```

下面的程序代码则展示了如何通过 **MultipleTransactionsSupported** 特性值来检查数据源是否支持嵌套事务。如果数据源支持嵌套事务，那么就调用 **TSQLConnection** 组件的 **StartTransaction** 并且通过指定不同的事务 ID 来激活嵌套事务：

```
const
    cnOuterTranID = 1;
    cnInnerTranID = 2;

procedure TForm1.Button1Click (Sender: TObject);
var
    outerTran : TTransactionDesc;
    innerTran : TTransactionDesc;
begin
    if (Self.SQLConnection1.MultipleTransactionsSupported) then
    begin
        outerTran.TransactionID := cnOuterTranID;
        outerTran.IsolationLevel := xilREADCOMMITTED;
        Self.SQLConnection1.StartTransaction (outerTran);
        try
            innerTran.TransactionID := innerTran;
            innerTran.IsolationLevel := xilREADCOMMITTED;
            Self.SQLConnection1.StartTransaction (innerTran);
            try
                //执行数据库相关工作程序代码
            ...
            Self.SQLConnection1.Commit (innerTran);
        except
            Self.SQLConnection1.Rollback (innerTran);
        end;
        //执行数据库相关工作程序代码
    ...
        Self.SQLConnection1.Commit (outerTran);
    except
        Self.SQLConnection1.Rollback (outerTran);
    end;
end;
Self.SQLConnection1.StartTransaction (TTransactionDesc);
end;
```

从上面的讨论中可以知道 dbExpress 对于事务的支持是非常齐全的，并且在使用上也非常方便。不过程序员要想充分掌握事务的功能，仍然必须彻底了解事务中隔离性（Isolation）代表的意思。这就是下一小节讨论的主题。

5.2 数据库的 Trans so ation

TransIsolation 对于事务有很重要的影响，因此程序员在使用 dbExpress 事务时必须了解 TransIsolation 的意义，如此才能够保证数据的一致性以及正确性。不过在解释 dbExpress 支持的 TransIsolation 之前，了解在多人使用的数据库应用系统中数据可能处于的状态以及意义，可以帮助程序员很快地了解为什么要使用 TransIsolation。

一般来说，在一个多人使用的数据库应用系统中，数据可能会同时被许多客户端的用户处理。例如目前可能有多个用户在处理员工数据表中的数据。那么这些数据可能会有如下的状态：

- Dirty Read
- Nonrepeatable Read
- Phantom

所谓 Dirty Read 状态是指一个客户端在激活事务之后，会读取其他事务已经修改过但是尚未 Commit 回数据库的数据。例如：

- 1) 一个客户端用户使用 dbExpress 激活事务，这个事务试着修改 R&D 部门中 Delphi/Kylix 程序员的薪资数据。
- 2) 此时，另外一个客户端用户也激活一个事务，开始查询 R&D 部门中 Delphi/Kylix 程序员的薪资数据。
- 3) 第一个客户端的用户决定取消对于 R&D 部门中 Delphi/Kylix 程序员的薪资数据的修改动作，因此调用了 TSQLConnection 的 Rollback 方法。

在上面的步骤 3 发生之前，在 Dirty Read 状态下第二个用户查询到了第一个用户已经修改但是没有 Commit 过的数据。稍后，第一个用户取消了对于数据的修改，因此第二个用户查询到了不存在的数据。

而 Nonrepeatable read 状态是指在一个事务中会取得不一致的数据，不过这个状态可以正确地取得其他事务 Commit 的数据。让我们再使用刚才的场景来解释这个意思：

- 1) 用户 A 激活了一个 2 阶段的事务，在第 1 阶段中他查询 R&D 部门中 Delphi/Kylix 程序员的薪资数据。
- 2) 用户 B 激活并且完成了一个事务，在这个事务中用户 B 修改了 R&D 部门中 Delphi/Kylix 程序员的薪资数据。
- 3) 用户 A 完成了他的事务，并且在这个事务的第 2 阶段中用户 A 再度查询 R&D 部门中 Delphi/Kylix 程序员的薪资数据，以计算 Delphi/Kylix 程序员的平均薪资。

如果没有设置正确的 **TransIsolation** 模式，那么用户 A 将会在两次读取中得到不同的薪资数据。因此这个读取的结果是 **Nonrepeatable read**。

最后的 **Phantom** 是 **Nonrepeatable read** 的一个变体，这个状态的意义如下：

1) 用户 A 激活了一个事务，他查询 R&D 部门中 Delphi/Kylix 程序员的薪资数据。

2) 用户 B 激活并且完成了一个事务，这个事务在 R&D 部门中新增了一个 Delphi/Kylix 程序员的记录。

3) 用户 A 完成了他的事务，并且在这个事务的最后用户 A 再度查询 R&D 部门中 Delphi/Kylix 程序员的薪资数据，以计算 Delphi/Kylix 程序员的平均薪资。

在上面的情形中当用户 A 执行事务之后，他在步骤 3 中会读取到一个新的 Delphi/Kylix 程序员记录，但是这个记录在先前的读取中并没有出现，因此这个新记录就如同幻影一样突然出现。

现在读者应该了解了在数据库应用系统中数据可能处于的状态的意义。

dbExpress 支持的 **TransIsolation** 总结在下面的表格中。下面表格中的 **Serializable** 模式应该可以使用 **TTransactionDesc** 中的 **CustomIsolation** 来设置，不过由于目前 dbExpress 并没有实现 **CustomIsolation** 而且不是每一个关系型数据库都支持 **Serializable** 事务模式，因此在目前的 dbExpress 版本中无法使用 **Serializable** 模式的 **TransIsolation** 设置。

TransIsolation	意 义
DirtyRead	这个程度的 TransIsolation 代表目前的事务可以看得其他事务对于数据的修改，即使其他事务尚未 Commit
ReadCommitted	这个程度的 TransIsolation 代表目前的事务只看得其他事务 Commit 的修改数据。但是这种模式可能会看到不一致的数据
RepeatableRead	这个程度的 TransIsolation 保证可看到一致性的数据。因为在这种模式中能够看到的数据是在目前的事务激活之前已被其他事务 Commit 的数据
Serializable	最严格的 TransIsolation 模式，代表数据库中所有事务是一个接一个地执行，每一个事务看不到其他事务的任何影响

现在读者就可以了解如何设置 **TTransactionDesc** 中的 **TransIsolation** 了，下面的表格列出了 dbExpress 中能够为 **TTransactionDesc** 的 **IsolationLevel** 字段设置的值以及这些设置值的意义。

IsolationLevel	意 义
xiLDIRTYREAD	代表使用 DirtyRead 的 TransIsolation 程度
xiLREADCOMMITTED	代表使用 ReadCommitted 的 TransIsolation 程度
xiLREPEATABLEREAD	代表使用 RepeatableRead 的 TransIsolation 程度
xiLCUSTOM	代表使用定制的 TransIsolation 程度

例如，如果程序员设置 **TTransactionDesc.IsolationLevel** 为 **xiLREADCOMMITTED**，

那么就是使用 ReadCommitted 模式，就可能会取得 Nonrepeatable read 和 Phantom 的数据。如果程序员设置 TTransactionDesc.IsolationLevel 为 xilREPEATABLE_READ，那么就可保证不会读取 Nonrepeatable read 的数据，但是仍然有可能读取 Phantom 的数据。

下面的表格完整地总结了每一种 TransIsolation 设置能够读取到的数据种类：

	Dirty Read	Nonrepeatable Read	Phantom
Read UnCommitted (即 Dirty Read)	是	是	是
Read Committed	否	是	是
Repeatable Read	否	否	是
Serializable	否	否	否

dbExpress 驱动程序在默认情况下使用 Read Committed 的 TransIsolation 设置，这是一个很适合的默认设置，读者如果没有特别的需要可以接受这个默认设置。如果读者决定使用其他的 TransIsolation 设置值，那么一定要真正了解为什么需要使用其他的 TransIsolation 设置，因为这将会影响数据的完整性以及 dbExpress 应用程序的性能。

如果读者想要进一步了解事务的原理，那么可以参考由 Philip A. Bernstein 和 Eric Newcomer 所著的《Principles Of Transaction Processing》。

5.3 错误处理

在程序员开发 dbExpress 应用系统时，对于数据修改产生的错误应该特别注意，如果程序员在错误事件发生时不处理这些错误的话，那么可能会造成数据流失。此外在 dbExpress 应用系统执行时，如果数据修改产生了错误，那么这些错误可能是违反了后端数据库定义的数据限制 (constraint)，也可能是违反数据表的索引定义等的状况。在这种情形下，程序员不但需要处理错误，也需要知道到底是什么发生了错误。例如许多程序员希望能够知道发生错误时后端数据库的原始错误码，以便将它转换成适当的中文消息通知给用户。

程序员要想能够在 dbExpress 应用系统中正确地处理错误以及取得真正错误的源代码的话，必须对于 Delphi/Kylix 如何触发错误处理函数有非常清楚的认识，这样才能毫无困难地处理任何错误以及访问任何错误消息。Delphi/Kylix 的 dbExpress 应用程序在调用 TSimpleDataSet/TClientDataSet 的 ApplyUpdates 方法修改数据时，如果发生了任何错误，那么 DataSnap 首先会触发应用程序服务器中 TDataSetProvider 组件的 OnUpdateError 事件处理函数，让应用程序服务器有机会处理错误，然后 dbExpress 会触发客户端应用程序中 TClientDataSet 组件的 OnReconcileError 事件处理函数让客户端应用程序有机会处理错误。图 5-1 描述了这个过程。

如果程序员是使用 Delphi 7 的 TSimpleDataSet 组件，那么 dbExpress 会直接触发应用程序中 TSimpleDataSet 组件的 OnReconcileError 事件处理函数让客户端应用程序有

机会处理错误。

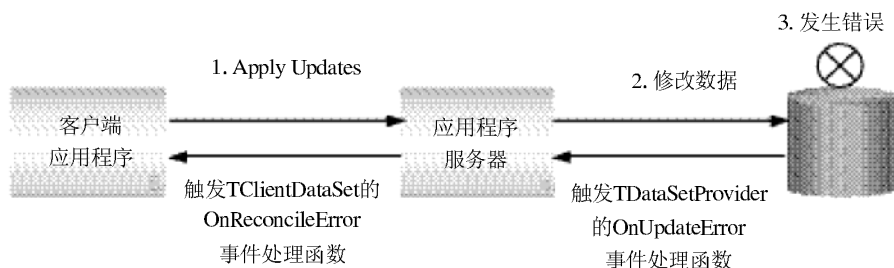


图5-1 dbExpress应用系统发生错误时触发的事件处理函数

在讨论处理错误的程序代码技术之前，先让我们看看一些 dbExpress应用程序在修改数据时可能发生的错误，以及应用程序应该如何处理这些情形。图 5 2是一个 dbExpress应用程序在处理数据时可能发生的问题。由于在 dbExpress应用程序中，在客户端应用程序调用 TSimpleDataSet/TClientDataSet的ApplyUpdates方法之前，其他用户有可能修改了数据库中相同的记录。发生这个情形的原因除了是因为 Delphi/Kylix使用乐观的锁定机制之外，也与 dbExpress把所有数据暂时存储在客户端的缓存内存中有关。例如在图 5 2中当用户 A 修改了“李维”这个记录，把李维的 OCCUPATION 字段值从产品经理改为行销经理，然后调用 TSimpleDataSet/TClientDataSet的ApplyUpdates想把这个记录更新回数据库中。但是在用户 A 调用 ApplyUpdates之前，用户 B 已经修改了同一个记录。用户 B 把李维的 OCCUPATION 字段值从产品经理改为技术经理。此时当用户 A 的 dbExpress 要更新数据时，它会在数据表中找寻一个“李维，34，产品经理”的记录，由于这个记录已经被用户 B 改变了所以用户 A 的 dbExpress 无法在数据表中找到这个记录，因此会发生错误。此时 dbExpress 和 DataSnap 会触发 TDataSetProvider 的 OnUpdateError 事件处理函数，最后再触发客户端应用程序中 TClientDataSet 的 OnReconcileError 事件处理函数，或是直接触发 TSimpleDataSet 的 OnReconcileError 事件处理函数。当这个情形发生时，请问读者应该怎么办？由于这个时候我们不知道李维到底是行销经理还是技术经理，所以在这种情形中 DataSnap 应用程序应该把这些冲突的数据显示给用户看，然后要求用户采取相对应的行动。

但是，如果当用户 A 调用 ApplyUpdates 方法时发生了图 5 3 的情形，那么用户 B 虽然修改了李维的 AGE 字段值而造成错误，但是因为用户 A 是修改李维的 OCCUPATION 字段值，与 AGE 字段没有冲突，所以在这种情形中 dbExpress 应用程序可以自行决定修正这个错误，接着再次把数据写入数据表中。例如，用户 A 的应用程序可以接受数据表中李维最新的 AGE 字段值 35，然后把“李维，35，行销经理”的数据更新回数据表中。当然 dbExpress 应用程序也可以将这些信息显示给用户，要求用户 A 决定如何处理，或是在用户 A 决定不亲自处理时再由 dbExpress 自行修改这个

AGE字段的错误，然后再次将修改数据更新回数据表中。

从上面的讨论中可以了解到，在 dbExpress应用系统中发生错误时，程序员可以有数种不同的方式来处理错误，并没有绝对的答案。不过将冲突数据的数值和原因显示给用户，能够让用户根据这些信息做出正确的响应行动，这应该还是比较适当的。

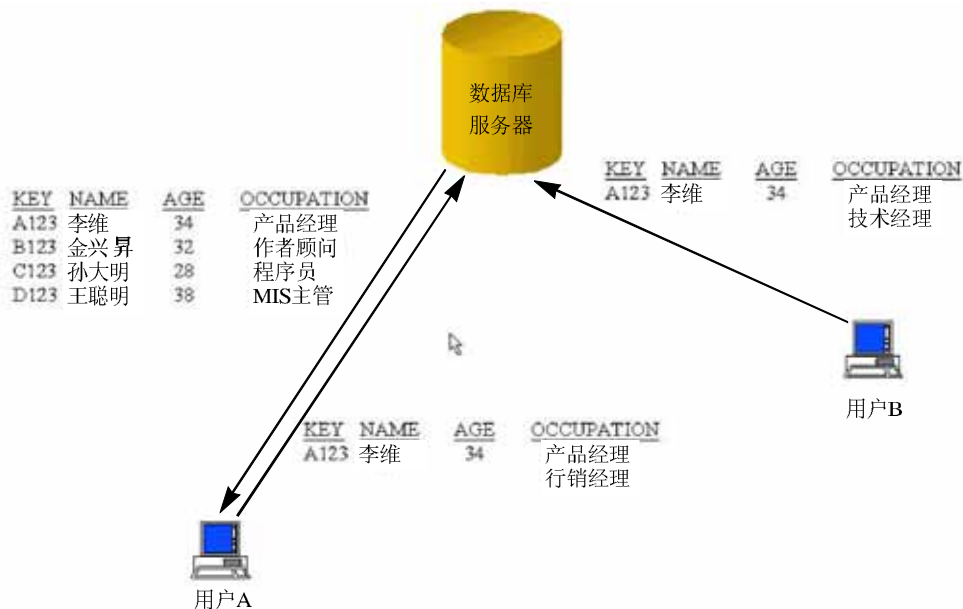


图5-2 dbExpress应用程序可能发生的错误情形

当发生错误时，DataSnap在触发 TDataSetProvider的OnUpdateError和触发 TSimpleDataSet/TClientDataSet的OnReconcileError事件处理函数时传递的错误对象的类型是不一样的。图 5 4说明了 OnUpdateError和OnReconcileError事件处理函数接受的错误对象的类类型。

程序员可以在这两个事件处理函数中通过这个传入的错误对象进行非常多的工作。例如取得错误消息文本，或是取得后端数据库产生的原始错误码。图 5 5是 EReconcileError和EUpdateError这两个错误对象的继承图，从图中可以看到 EReconcileError错误对象和EUpdateError错误对象都是从EDatabaseError对象继承下来的，所以 Delphi/Kylix程序员可以通过类型转换从它的父类中取得需要的特性值，在稍后的实现小节中会有范例说明如何使用 Object Pascal来解决这些问题。

首先让我们看看 TDataSetProvider组件的OnUpdateError事件处理函数的程序原型：

```
procedure TForm1.DataSetProvider1UpdateError(Sender: TObject;
  DataSet: TCustomClientDataSetE; EUpdateError: TUpdateError; UpdateKind: TUpdateKind; var
  Response: TResolverResponse
begin
end;
```

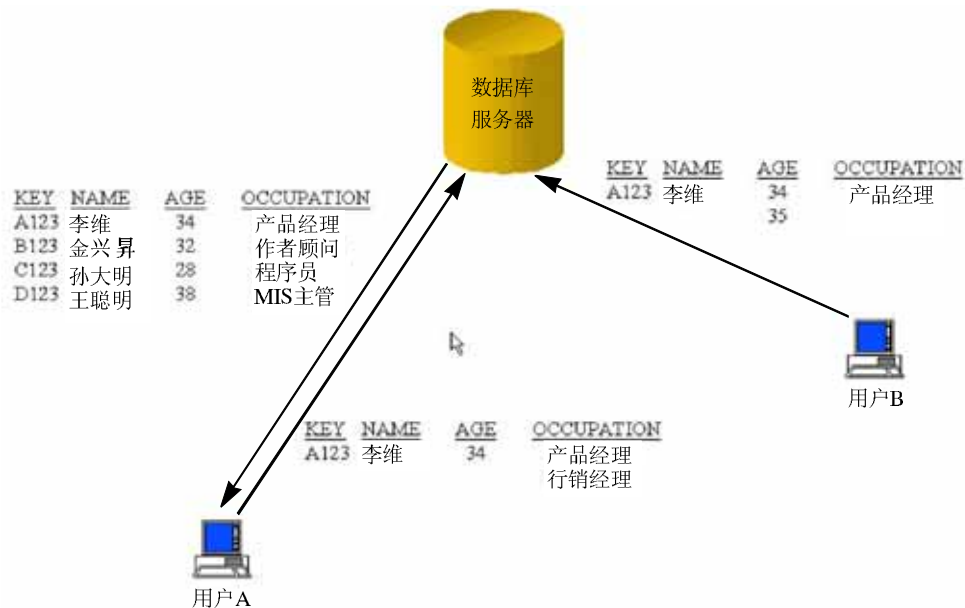



图5-3 dbExpress应用程序可能发生的错误情形



图5-4 错误事件处理函数接受的错误对象

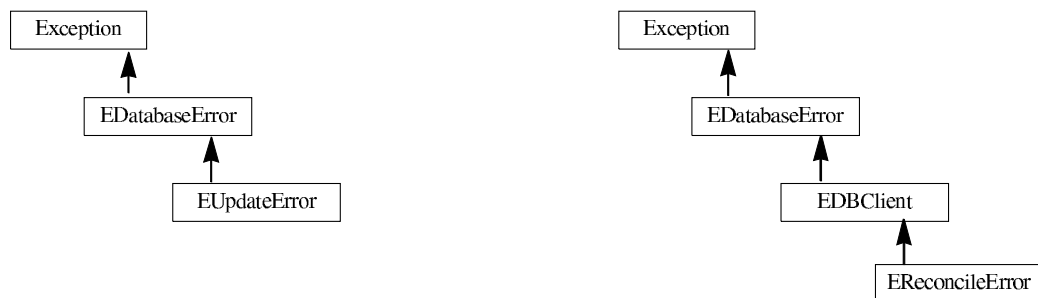


图5-5 EUpdateError和EReconcileError对象的继承图

在OnUpdateError事件处理函数中接受四个参数，第一个参数 **Sender**代表触发这个事件处理函数的对象，第二个参数 **DataSet**代表发生错误的数据集组件，而且 **DataSet**中目前的记录就是发生错误的数据库，第三个参数 **E**是发生错误时Delphi/Kylix产生的异常对象。程序员可以从这个对象中取得所有详细的错误信息，例如发生错误的原因、错误的原始数据库错误码等。 **UpdateKind**代表是什么修改行为造成了错误，例如是新增数据、修改数据、或是删除数据的动作。最后一个参数 **Response**是非常重要的参数，这个参数必须由程序员使用 **Object Pascal**程序代码设置，当程序员

设置 **Response** 参数值并且 **OnUpdateError** 事件处理函数执行完毕之后，**dbExpress/DataSnap** 会根据程序员设置的值来决定如何继续处理这个错误。

首先让我们讨论一下 **OnUpdateError** 中的 **EUpdateError**。这个对象代表发生错误的原因以及数据库产生的原始错误码，程序员可以通过它了解有关错误的信息，进而根据这些信息来决定如何处理错误。**EUpdateError** 类有如下的特性：

特 性	意 义
Context	错误堆栈的内容信息
ErrorCode	由 dbExpress/BDE/ADO 返回的错误码
OriginalException	代表这个修改错误的异常对象
PreviousError	前一次修改行动的错误码

在这些特性中，程序员除了可以通过 **Context** 取得错误的消息之外，最重要的就是 **ErrorCode** 这个特性。**ErrorCode** 包含 **dbExpress/BDE/ADO** 返回的错误码，程序员可以通过判断 **ErrorCode** 是不是 0 来判断这个错误是否是由 **dbExpress** 产生的。如果 **ErrorCode** 是 0，那么程序员可以通过 **OriginalException** 对象来取得进一步的错误信息。

Response 参数则是 **OnUpdateError** 事件处理函数中最重要的参数，因为 **dbExpress/DataSnap** 接下来如何处理错误完全是根据程序员所设置的 **Response** 值来决定的。下面的表格列出了 **Response** 可以设置的值以及这些值代表的意义。

值	意 义
rrSkip	跳过这个产生错误的记录，并且把这个记录继续留在缓存内存中
rrAbort	中断整个更新的行动，并且 Rollback 所有修改的数据
rrMerge	把数据封包中修改的数据和数据表中的数据互相合并
rrApply	这个设置值代表程序员在 OnUpdateData 事件处理函数中已经修正了错误的原因，而要求 Delphi/Kylix 再更新数据一次
rrIgnore	忽略产生错误的这个记录，而且也不会再把这个产生错误的记录返回给客户端应用程序，让客户端应用程序有机会再进行额外的处理

例如在前面讨论修改李维数据的范例中，如果发生错误的原因是因为另外一位用户改变了李维的 **AGE** 字段值，那么用户如果认为另外一位用户修改的 **AGE** 字段和我们修改的 **OCCUPATION** 字段没有冲突，那么在 **OnUpdateError** 事件处理函数中程序员可以重新设置李维的 **AGE** 字段为 35，然后设置 **Response** 为 **rrApply** 要求 **dbExpress/DataSnap** 再更新一次。反之，如果另外一位用户把李维的 **OCCUPATION** 字段改为销售经理，那么就与我们修改的数据有冲突，此时程序员可以设置 **Response** 为 **rrSkip** 先跳过这个修改的记录，然后让用户在客户端应用程序的 **OnReconcileError** 事件处理函数中再根据应用程序的情况决定如何继续处理。在本小节实际开发一个范例之前，先介绍 **TSimpleDataSet/TClientDataset** 的 **OnReconcileError** 事件处理函数。下面是这个事件处理函数的原型：

```
procedure TForm3.ClientDataSet1ReconcileError (DataSet: TCustomClientDataSet;  
    E: EReconcileError; UpdateKind: TUpdateKind;  
    var Action: TReconcileAction;  
begin  
end;
```

OnReconcileError事件处理函数和前面介绍的 OnUpdateError事件处理函数非常类似，它们的参数意义也非常接近。其中 Action代表程序员可以采取的行动，它有如下的值：

值	意 义
raSkip	跳过这笔记录，并且把用户对于这个记录修改的值保留在 TSimpleDataSet/TClientDataSet的 Delta中
raAbort	中止整个更新数据的流程，Rollback所有修改
raMerge	把这个需要修改的记录和目前数据表中的数据合并
raCorrect	用在事件处理函数中指定的新值更正目前需要更新的数据
raCancel	取消对于这个记录的所有修改，并且恢复所有字段的旧值
raRefresh	取消对于这个记录的所有修改，并且以目前数据表中的字段值来代替这个记录的值

程序员可以通过在 OnReconcileError事件处理函数中设置 Action的值来处理错误。当程序员在 OnUpdateError或是 OnReconcileError事件处理函数中处理错误时，可以使用字段对象的以下四个特性值来取得目前记录的修改数据值（Value）、字段在修改之前的旧值（OldValue）、现在要设置的新字段值（NewValue）以及目前在数据表字段中最新的字段值（CurValue）。

值	意 义
Value	目前修改过的字段值
OldValue	代表字段在修改之前的值
NewValue	代表程序员在 OnUpdateError或是 OnReconcileError事件处理函数中设置的新字段值
CurValue	代表目前存在于数据源中数据表的字段值

而下面的表格则是 EReconcileError异常对象的特性以及它们的意义。

特 性	意 义
Context	错误堆栈的内容信息
ErrorCode	由dbExpress/BDE/ADO返回的错误码

有了这些基本但是重要的概念之后，现在就可以让我们看几个范例程序来说明如何运用前面介绍的概念处理错误情形。

在下面的范例中，我们将使用 dbExpress连接 MS SQL Server pubs 数据库中的 authors 数据表。然后我们修改其中一个记录的 au_id 字段的值而且故意把这个值改成和其他记录一样，再调用 TClientDataSet 的 ApplyUpdates 更新回 MS SQL Server 中。

由于au id字段是索引字段,因此如此一来会造成 MS SQL Server产生键值冲突的错误,dbExpress便会触发TDataSetProvider的OnUpdateError以及TClientDataSet的OnReconcileError事件处理函数。

下面是TDataSetProvider的OnUpdateError事件处理函数以及 TClientDataSet的OnReconcileError事件处理函数的实现程序代码。在这些实现程序代码中显示错误的错误码以及错误消息。此外在 TClientDataSet的OnReconcileError事件处理函数中还根据用户在范例程序发生错误时是设置取消更新或是中止整个更新的流程的选项来设置TReconcileAction对象的数值。

```
procedure TdmHandleError.dspAuthorsUpdateError (Sender: TObject;  
    DataSet: TCustomClientDataSet; E: EUpdateError; UpdateKind: TUpdateKind;  
    var Response: TResolverResponse  
begin  
    frmdbxError.SetErrorcode (E.ErrorCode);  
    frmdbxError.SetContext (E.Context);  
    if (E.ErrorCode <> 0) then  
    begin  
        frmdbxError.SetErrorMessage (E.OriginalException.Message);  
    end;  
end;  
  
procedure TdmHandleError.cdsAuthorsReconcileError (  
    DataSet: TCustomClientDataSet; E: EReconcileError;  
    UpdateKind: TUpdateKind; var Action: TReconcileAction  
begin  
    if (frmdbxError.cbCancelError.Checked) then  
        Action := raCancel;  
    if (frmdbxError.cbAbort.Checked) then  
        Action := raAbort;  
end;
```

现在让我们直接观察范例程序对于错误的反应。图 5 6是此范例程序刚执行时的画面。

接着我们故意修改 au id字段值,让第一个记录的数值和第 3个相同,然后点击主窗体中的ApplyUpdates按钮把修改过的数据更新回MS SQL Server中(见图5 7)。

由于这个更新的动作一定会让 MS SQL Server产生错误,因此 dbExpress也在TDataSetProvider的OnUpdateError事件处理函数中拦截到错误(见图5 8)。

请读者注意,在图5 8中当dbExpress拦截到错误时,下方的两个TDBGrid显示了调用ApplyUpdates之前以及之后的 Delta数值。由于我们没有在 TSimpleDataSet/TClientDataSet的OnReconcileError中指定任何值给 TReconcileAction对象,因此即使发生了错误,dbExpress仍然会保存尚未成功更新回数据源的Delta值,这可以让dbExpress

应用程序在修正了错误原因之后有机会再次把Delta中的数据更新回数据源中。

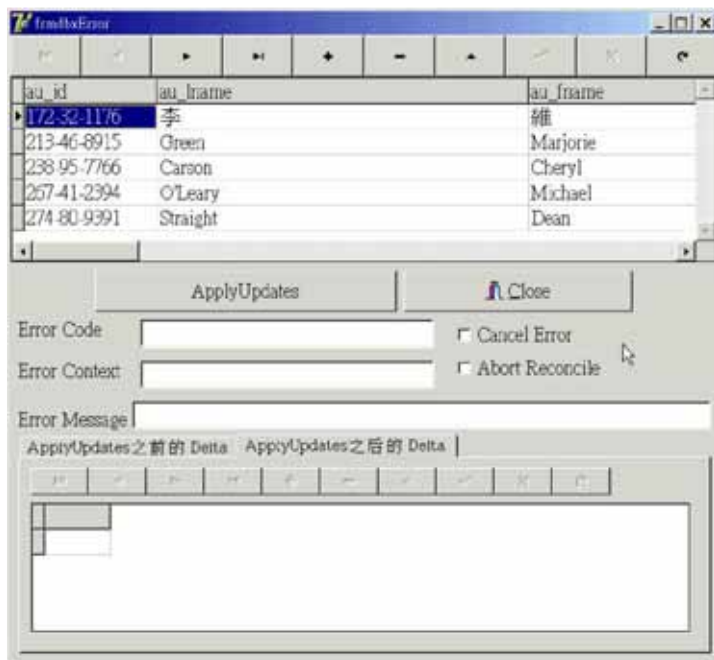


图5-6 范例程序刚执行时的画面

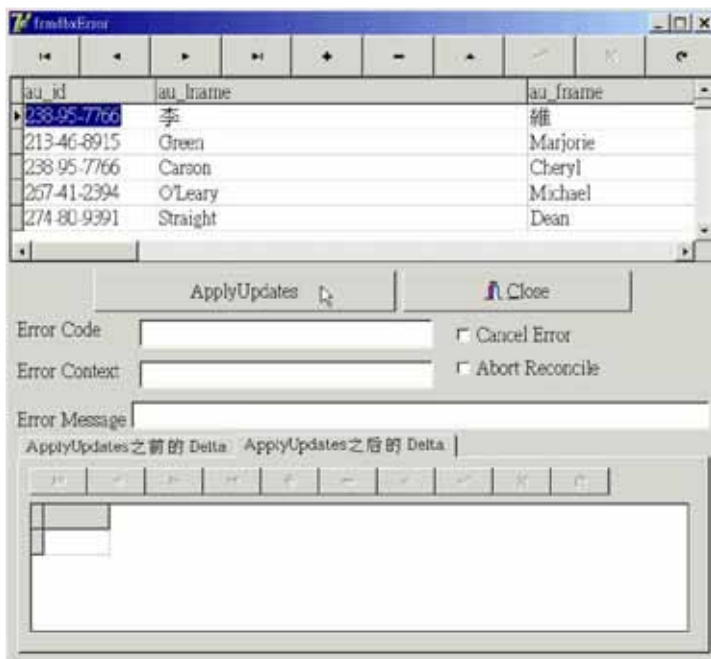


图5-7 故意修改au id字段值，让第一个记录的数值和第3个相同

当然，如果程序员在 TSimpleDataSet/TCClientDataSet 的 OnReconcileError 中将 rrCancel 指定给 TReconcileAction 对象，那么就代表要取消这个更新行动，那么 dbExpress 便会清除 Delta 中的数值，并且把数据恢复成原始的数值。例如在图 5 9 中我们在范例程序的主窗体中先设置 Cancel Error 选项，那么在点击了 ApplyUpdates 按钮之后下方显示的 ApplyUpdate 之后的 Delta 数值已经被 dbExpress 自动清除了，这与图 5 8 显示的执行行为是不一样的。

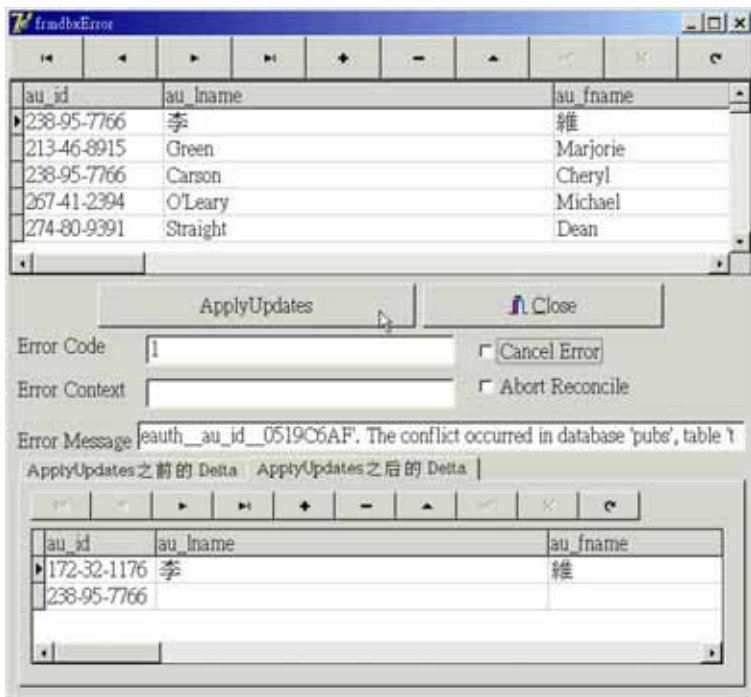


图5-8 dbExpress拦截到错误并且显示错误消息

很幸运的是，如果程序员需要在 TSimpleDataSet/TCClientDataSet 的 OnReconcileError 事件处理函数中处理错误的话，那么程序员不需要编写复杂的程序代码，就能够从 Delphi/Kylix 产生的错误对象中取出各种信息，因为 Delphi/Kylix 已经帮助程序员编写好了这些程序代码。在 Delphi/Kylix 的 New Items 对话框的 Dialogs 选项卡中有如图 5 10 所示的 CLX Reconcile Error Dialog。程序员可以点击这个图标以产生一个可以在 TSimpleDataSet/TCClientDataSet 的 OnReconcileError 事件处理函数中调用并且处理错误的对话框。

现在就让我们使用这个对话框在范例程序中处理范例程序发生的错误。首先按照图 5 10 在客户端应用程序中建立这个 Reconcile Error 对话框，然后在 TSimpleDataSet/TCClientDataSet 的 OnReconcileError 事件处理函数中加入如下的程序代码：

```
procedure TForm1.ClientDataSet1ReconcileError (DataSet: TClientDataSet;
```



```
E: EReconcileError; UpdateKind: TUpdateKind;  
var Action: TReconcileAction;  
begin  
    HandleReconcileError (Dataset, UpdateKind, E)  
end;
```

HandleReconcileError方法是Delphi/Kylix的Reconcile Error对话框提供的方法，程序员只需要在 TSimpleDataSet/TClientDataSet的OnReconcileError事件处理函数中调用它即可。此外要使用这个对话框，程序员还必须做另外的两个动作，需要在客户端应用程序中uses这个对话框。

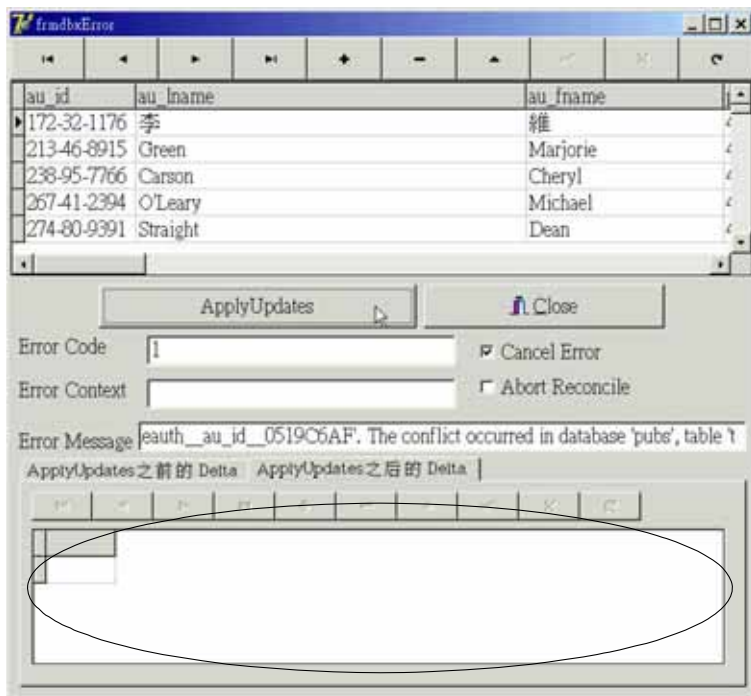


图5-9 如果选择主窗体中的Cancel Error，那么Delta会被清除



图5-10 Delphi/Kylix已经提供的Reconcile Error对话框

现在，客户端应用程序就可以使用这个对话框来处理错误了。让我们再执行一次

范例程序，然后同样修改 `au_id` 字段的值，那么此时客户端应用程序便会显示 **Reconcile Error** 对话框，如图 5-11 所示。

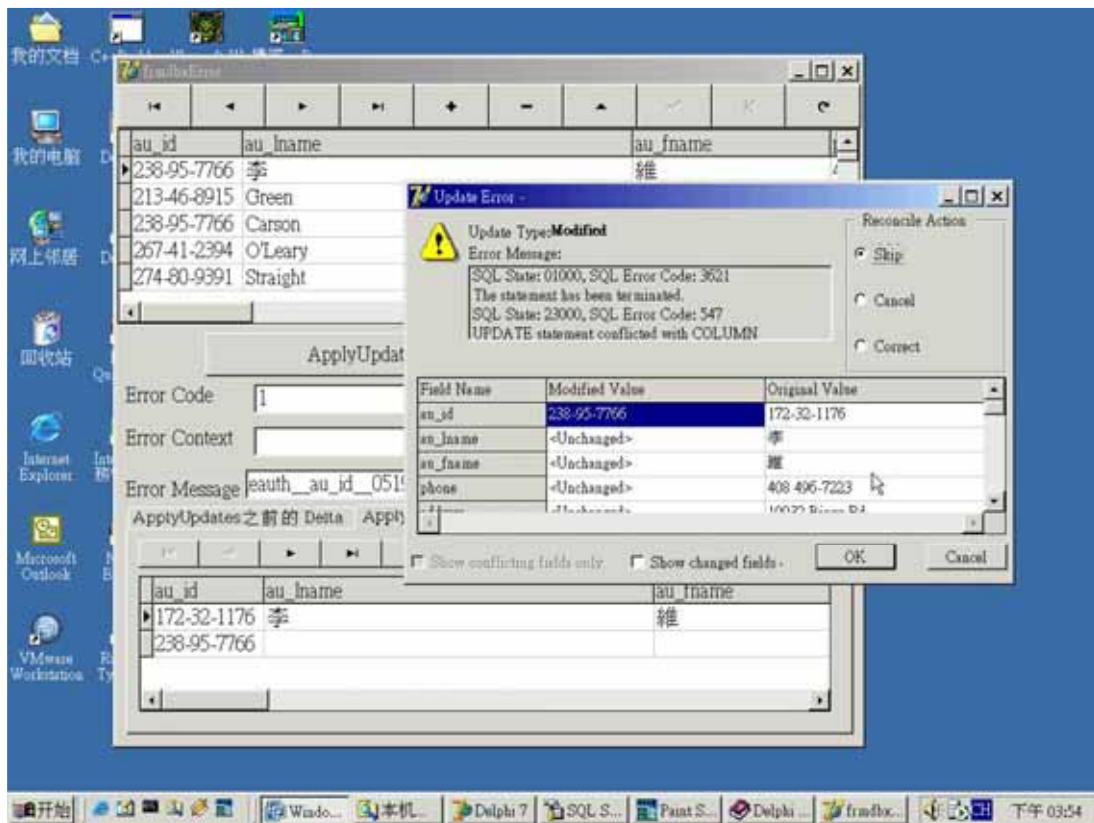


图5-11 Reconcile Error对话框显示详细的错误消息

在 **Reconcile Error** 对话框中会显示详细的错误消息以及发生错误的字段的新旧值，而且在对话框的右边显示了程序员能够采取的行动。

如果现在我们要取消这个更新操作，那么可以点击 **Reconcile Error** 对话框右边的 **Cancel** 选项。请读者注意，一旦我们点击了 **Cancel** 选项，那么原先被修改的 `au_id` 字段值马上会被恢复成 `<Unchanged>`，如图 5-12 所示。这代表 **dbExpress** 会清除用户对于 `au_id` 字段的修改并且把 `au_id` 的值恢复成原始的数值。

事实上，**Reconcile Error** 对话框之所以能够显示发生错误的字段的新旧值是因为它使用前面介绍的 **TField** 的 **OldValue** 和 **Value** 的特性值，而程序员如果选择更正错误的话，就可以设置 **TField** 的 **NewValue** 特性来要求 **dbExpress** 试着更正错误。**Reconcile Error** 对话框有许多精彩的程序代码，有兴趣的读者可以参考一下，读者甚至可以试着修改它让 **TDataSetProvider** 的 **OnUpdateError** 事件处理函数也可以使用它。

希望本小节讨论的错误处理概念和技术能够让读者完全掌握 **dbExpress** 提供的错

误处理机制，并且能够正确地在读者的应用程序中使用。如果读者还没有在应用程序中使用任何错误处理程序代码的话，也希望读者阅读完本小节之后能够了解处理错误的重要性以及如何处理错误。

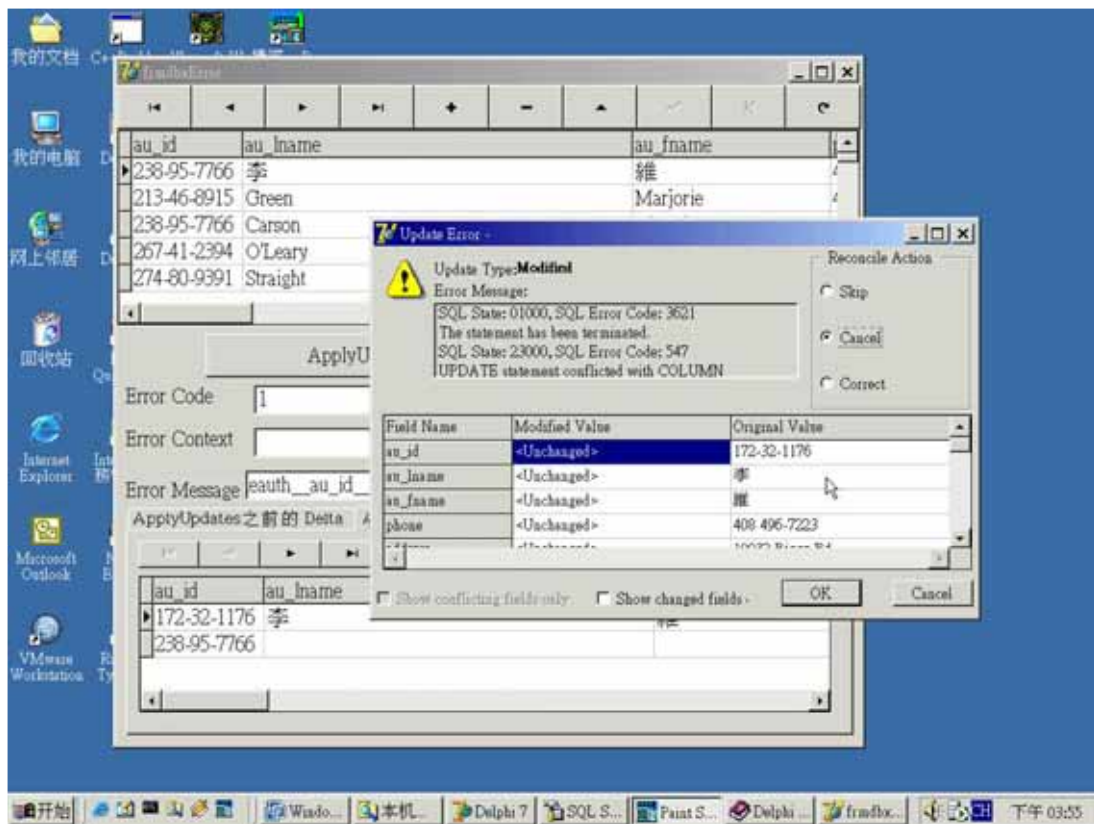


图5-12 选择取消发生的错误

5.4 在COM+中使用dbExpress

由于dbExpress For MS SQL Server是在OLE DB之上实现的，因此这代表dbExpress For MS SQL Server应该也可以在COM+中使用。由于dbExpress本身的精简和有效率，再配合COM+提供的各种缓冲池（Pooling）和缓存机制，这两者搭配起来是很合适的。本小节讨论的内容就是展示如何在COM+中使用dbExpress，读者从本小节中可以学习到许多在COM+中使用dbExpress的技巧。在读者了解了这些技巧之后，笔者在本小节之后也会说明一些在COM+中使用dbExpress时应该注意的事情。

在下面的小节中，本章将直接使用一个范例来说明如何使用Delphi建立COM+对象并且使用dbExpress来访问数据。本书并不讨论COM+的底层技术细节，如果读者

想了解COM+技术，那么请读者参阅相关的COM+书籍。对于使用Kylix的读者而言可以阅读参考或是跳过本小节讨论的内容。

步骤1：使用Delphi建立COM+组件

首先让我们使用Delphi快速建立ActiveX项目，再在其中建立COM+组件。点击File New Other...菜单，在ActiveX选项卡中选择ActiveX Library图标以建立ActiveX项目。接着再在同一选项卡中选择Transactional Object图标以建立COM+组件，见图5-13。



图5-13 在ActiveX函数库中建立支持COM+的Transactional Object

接着在图5-14的New Transactional Object对话框中设置这个COM+组件的名称为Threading Model，并且设置COM+组件使用的事务模式为Supports transactions。

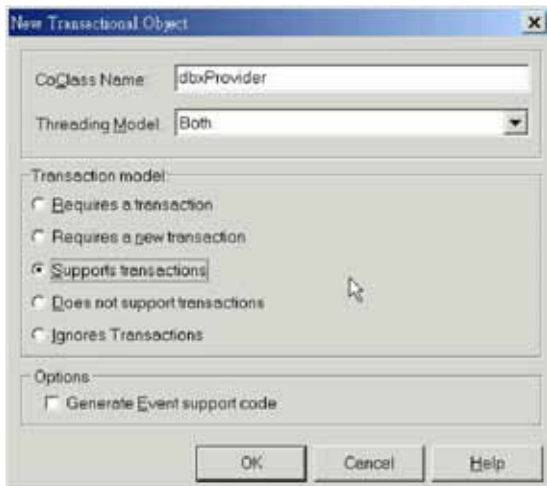


图5-14 设置Transactional Object支持COM+的事务功能

点击OK按钮之后，Delphi便会自动产生这个COM+组件的骨架程序代码。由于这个COM+组件将访问MS SQL Server中的数据，因此我们再在ActiveX项目中建立一个数据模块以便在其中使用dbExpress组件。请点击File New Data Module菜单建立数据模块，再在其中放入TSQLConnection、TSQLDataSet以及TSimpleDataSet组件，连接到MS SQL Server的pubs数据库，如图5 15所示。



图5-15 建立数据模块并且在其中放入dbExpress组件以连接MS SQL Server

数据模块中的TSimpleDataSet（sdsEmployee）在DataSet\CommandText特性值中使用了Select * from employee从pubs的employee数据表中取得数据。

完成了数据模块之后，现在我们就可以为COM+组件定义输出的方法以准备稍后让客户端的应用程序调用。要为COM+组件定义服务方法，请点击View Type Library菜单激活Delphi的Type Library编辑器（见图5 16），展开范例COM+组件的IdbxProvider接口，然后使用Type Library编辑器在COM+组件的接口中定义如下四个方法：getAllEmployees、getEmployeeByName、updateEmployee以及deleteEmployeeByName。这些方法分别允许客户端取得employee数据表的数据、更新employee数据表的数据以及删除employee数据表中特定的employee记录。

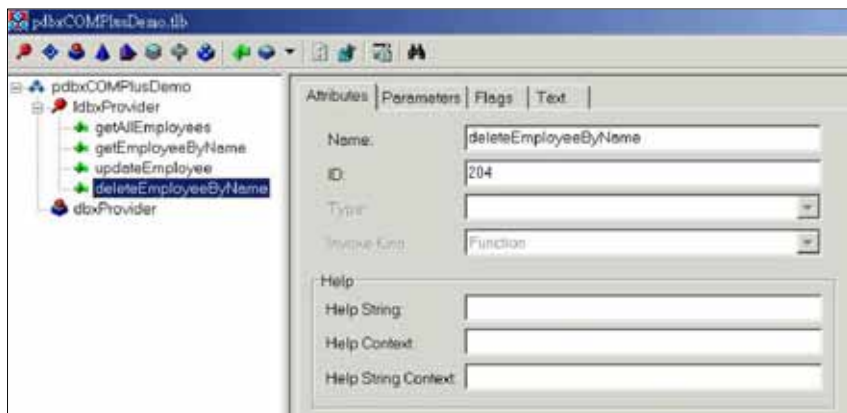


图5-16 在COM+组件的接口中定义输出的方法

使用Type Library编辑器定义服务方法之后，现在回到 Delphi的编辑器中准备开始实现这些方法。下面是这个范例 COM+组件的实现程序单元：

```

unit udbxProvider;
{$WARN SYMBOL_PLATFORM OFF}

interface

uses
    ActiveX, Mtsobj, Mtx, ComObj, pdbxCOMPlusDemo_TLB, StdVcl, SysUtils;

type
    TdbxProvider class (TMtsAutoObject, IdbxProvider
    protected
        procedure getAllEmployees (var vData: OleVariant; safecall);
        procedure deleteEmployeeByName (const sName: WideString; safecall);
        procedure getEmployeeByName (const sName: WideString;
            var vData: OleVariant; safecall);
        procedure updateEmployee (vData: OleVariant; safecall);

    end;

implementation

uses ComServ, udmdbxProvider;

const
    sAllEmployees = 'Select * from employee';
    sEmployeeByName = 'Select * from employee where fname = ' ;
    sDeleteEmployeeByName = 'delete employee where fname = :ID1';

procedure TdbxProvider.getAllEmployees (var vData: OleVariant;
begin
    try
        try
            dmdbxProvider := TdmdbxProvider.Create;
            dmdbxProvider.sdsEmployee.DataSet.CommandText := sAllEmployees;
            dmdbxProvider.sdsEmployee.Active := True;
            vData := dmdbxProvider.sdsEmployee.Data;
            SetComplete;
        except
            SetAbort;
        end;
    finally

```



```

    FreeAndNil (dmdbxProvider) ;
end;
end;

procedure TdbxProvider.deleteEmployeeByName (const sName: WideString)
begin
    try
        try
            dmdbxProvider := TdmdbxProvider.Create ;
            dmdbxProvider.sqldsGeneral.CommandText := sDeleteEmployeeByName;
            dmdbxProvider.sqldsGeneral.Params.ParamByName ('ID1').AsString := sName;
            dmdbxProvider.sqldsGeneral.ExecSQL (True) ;
            SetComplete;
        except
            SetAbort;
        end;
    finally
        FreeAndNil (dmdbxProvider) ;
    end;
end;

procedure TdbxProvider.getEmployeeByName (const sName: WideString;
    var vData: OleVariant);
begin
    try
        try
            dmdbxProvider := TdmdbxProvider.Create ;
            dmdbxProvider.sdsEmployee.Active := False;
            dmdbxProvider.sdsEmployee.DataSet.CommandText :=
                sEmployeeByName + ' ' + sName + ' ';
            dmdbxProvider.sdsEmployee.Active := True;
            vData := dmdbxProvider.sdsEmployee.Data;
            SetComplete;
        except
            SetAbort;
        end;
    finally
        FreeAndNil (dmdbxProvider) ;
    end;
end;

procedure TdbxProvider.updateEmployee (vData: OleVariant);
var

```

```

    aUpObj : IdBxUpProvider;
    sError : WideString;
begin
    try
        OleCheck (ObjectContext.CreateInstance (CLASS_dbxUpProvider, ID_IdBxUpProvider, aUpObj));
        aUpObj.UpdateEmployee (vData, sError);
        SetComplete;
    except
        SetAbort;
    end;
end;

initialization
    TAutoObjectFactory.Create (ComServer, TdbxProvider, Class_dbxProvider,
        ciMultiInstance, tmBoth);
end.

```

首先看看 `getAllEmployees` 方法是如何实现的，它的功能是从 `pubs` 的 `employee` 数据表中取得数据。`getAllEmployees` 先动态建立我们前面设计的数据模块，然后在 `sdsEmployee` 的 `DataSet\CommandText` 中指定 SQL 语句，开启 `sdsEmployee`，最后再把 `sdsEmployee` 取得的结果数据集作为 `OleVariant` 类型的返回结果传递给客户端。因此客户端应用程序只需要再把这个返回的 `OleVariant` 值指定给 `TSimpleDataSet` 或是 `TClientDataSet` 的 `Data`，就可以在客户端应用程序的数据感知组件中显示。

`getEmployeeByName` 方法的实现方式与 `getAllEmployees` 几乎是一样的，只是 `getEmployeeByName` 通过员工的名称组成 SQL 语句，再使用 `sdsEmployee` 从 `employee` 数据表中取得数据。

`deleteEmployeeByName` 允许客户端删除特定的 `employee` 记录，由于使用了 `delete` SQL 语句而 `delete` SQL 语句不返回结果数据集，因此我们可以直接使用数据模块中的 `TSQLDataSet` 来执行。`deleteEmployeeByName` 使用员工的名称组成 SQL 语句之后，就通过 `sqlDsGeneral` 的 `ExecSQL` 来执行这个 SQL 语句。

而 `updateEmployee` 方法允许客户端更新 `employee` 数据表的数据，`updateEmployee` 建立了另外一个负责更新数据的 COM+ 对象——`IdBxUpProvider`，稍后我们会讨论 `IdBxUpProvider` 对象的实现。由于 `IdBxUpProvider` 是 COM+ 对象，因此在 `TdbxProvider` 中是直接使用目前的 COM+ 对象的 `Object Context` 的 `CreateInstance` 方法来建立其他 COM+ 对象，这样可以让这些 COM+ 对象形成一个 COM+ 的 Action 路径，并且让目前 COM+ 对象的一些特性能够自动地传递给新建立的 `IdBxUpProvider` 对象。

现在请编译此范例 COM+ 组件，然后点击 `Run Install COM+ Objects` 菜单注册这个 COM+ 组件，如此一来我们就可以准备编写客户端应用程序来调用这个范例 COM+

对象了。

步骤2: 开发客户端应用程序

有了COM+组件之后实现客户端的应用程序就很简单了,它只是调用COM+对象提供的服务方法来处理数据。由于笔者在编写此客户端应用程序时是与刚才实现的COM+对象在同一台机器中,因此在客户端的应用程序中笔者直接在客户端的项目中加入了前面TdbxProvider对象的Type Library定义文件,并且在客户端应用程序的主窗体中使用了TdbxProvider对象的Type Library程序单元pdbxCOMPlusDemo.TLB。由于范例COM+组件和客户端应用程序在同一台机器中执行,因此范例客户端应用程序能够自动取得COM+对象的注册信息。如果读者是在不同的机器中执行,那么必须使用COM+服务建立COM+组件的客户端安装程序,再在客户端机器中执行COM+组件的客户端安装程序,之后才能在客户端建立和使用远程的范例COM+对象。

下面是本范例的客户端应用程序。当用户点击了“连接COM+对象”按钮之后,BitBtn1Click事件处理函数调用TdbxProvider对象的Type Library程序单元中的wrapper类的CreateRemote方法建立TdbxProvider对象并且将COM+对象的接口指定给dbxObj接口变量。CreateRemote方法接受的参数是COM+对象所在机器的名称。

接着用户如果点击“取得所有Employee数据”按钮,那么bbtnGetallEmployeeClick事件处理函数会使用dbxObj接口变量调用COM+对象提供的getAllEmployees方法以取得employee数据表中的所有数据,最后再把数据指定给主窗体中的TClientDataSet组件的Data特性值并且显示在数据感知组件中。

用户如果点击“取得特定Employee数据”按钮,那么bbtnGetEmployeeClick事件处理函数会调用COM+对象提供的getEmployeeByName方法,并且把用户在主窗体中输入的员工名称当成参数传递给getEmployeeByName以取得特定员工的数据,最后再把数据指定给主窗体中的TClientDataSet组件的Data特性值并且显示在数据感知组件中。

```
procedure TfrmMain.BitBtn1Click (Sender: TObject);
begin
    dbxObj := CodbxProvider.CreateRemote('C:\CORDONSERVER');
end;

procedure TfrmMain.bbtnGetallEmployeeClick (Sender: TObject);
var
    vData : OleVariant;
begin
    if not Assigned (dbxObj) then
        BitBtn1Click (Sender);

    if Assigned (dbxObj) then
```

```
begin
    dbxObj.getAllEmployees (vData);
    cdsData.Data := vData;
end;
end;

procedure TfrmMain.bbbtnGetEmployeeClick (Sender: TObject);
var
    vData : OleVariant;
begin
    if not Assigned (dbxObj) then
        BitBtn1Click (Sender);

    if Assigned (dbxObj) then
    begin
        dbxObj.getEmployeeByName (Edit1.Text, vData);
        cdsData.Data := vData;
    end;
end;
```

现在编译并且执行此客户端应用程序，并且点击主窗体中的“连接 COM+对象”按钮、“取得所有 Employee数据”按钮和“取得特定 Employee数据”按钮，那么读者应该会看到类似图 5-17 的画面，我们果然可以成功地使用 COM+对象和 dbExpress 来访问 MS SQL Server 中的数据，开发出了 Microsoft DNA 形式的应用系统。

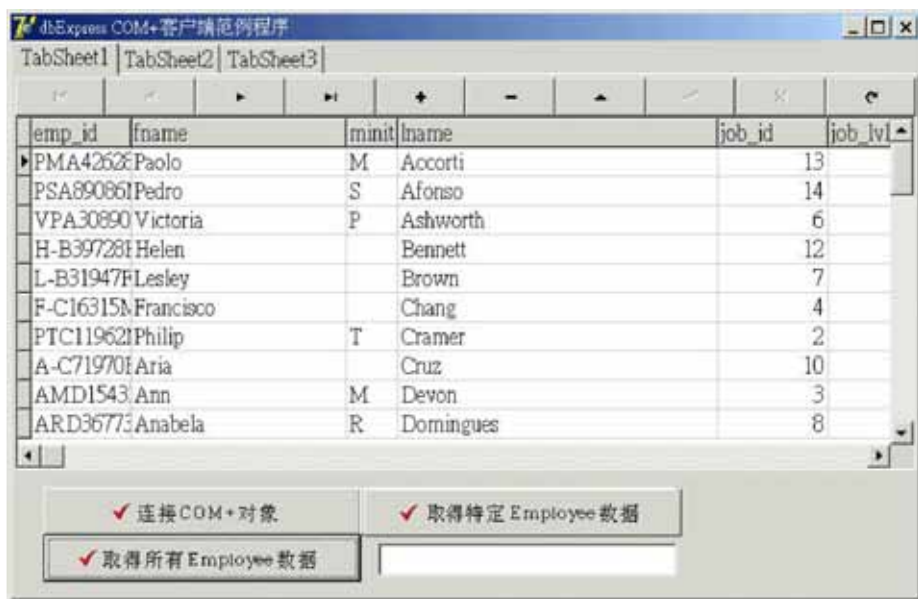


图5-17 使用范例客户端应用程序调用 COM+组件来访问数据

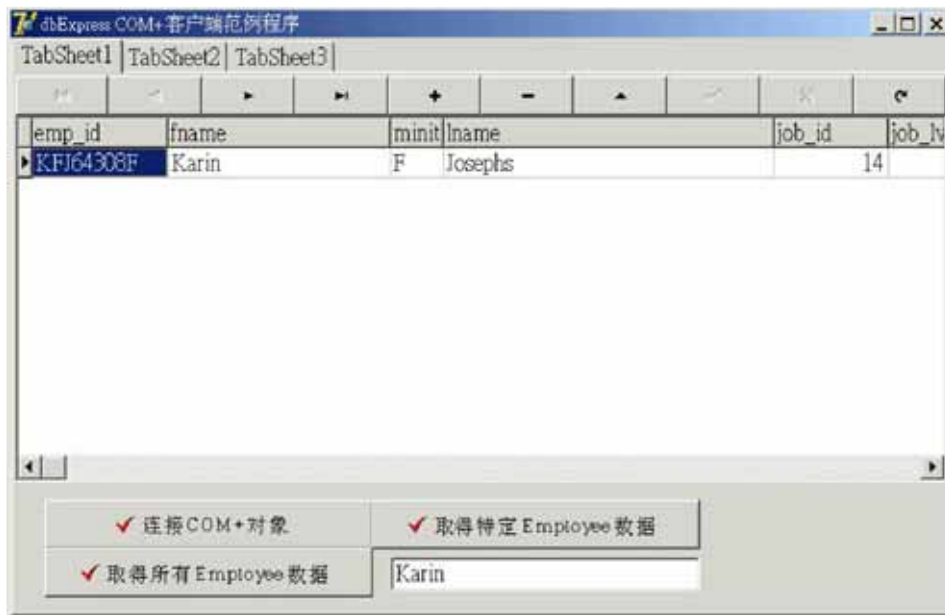


图5-17 (续)

这个范例尚未结束，因为我们仍然还没有实现更新数据的功能。由于前面的 TdbxProvider 对象提供查询数据的功能，因此让我们再开发另外一个组件来提供负责更新数据的 COM+ 对象。

回到 Delphi 中并且建立一个新的 ActiveX 项目，在其中建立一个 COM+ 对象，让我们为这个新的 COM+ 对象取名为 dbxUpProvider，并且设置它的 Threading Model 为 Both，Transaction Model 为 Requires a transaction，如图 5-18 所示。

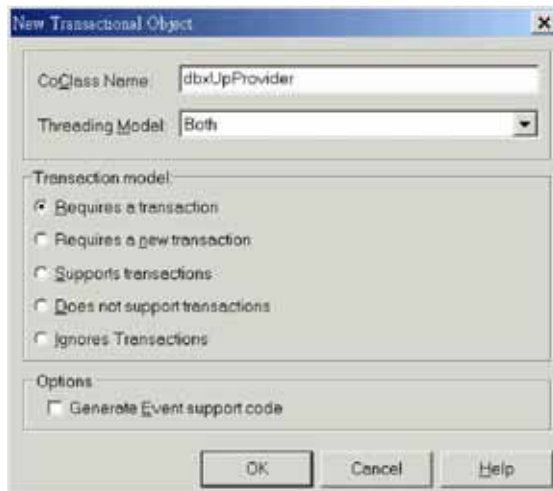


图5-18 建立可修改数据的COM+组件

接着激活Type Library编辑器，在这个COM+对象中定义一个UpdateEmployee方法以提供更新employee数据表的功能，如图5 19所示。

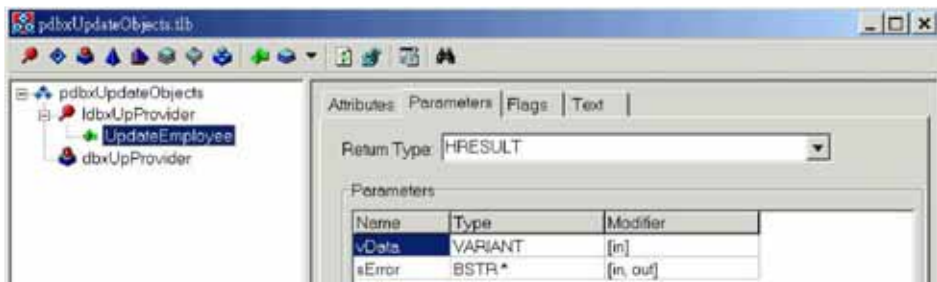


图5-19 在可修改数据的COM+组件中定义输出方法UpdateEmployee

由于dbxUpProvider同样需要使用dbExpress组件访问MS SQL Server，因此在这个ActiveX项目中也建立一个数据模块，并且放入 TSQLConnection以连接MS SQL Server，再放入一个TSQLDataSet组件准备执行更新数据的SQL语句，如图5 20所示。



图5-20 建立数据模块并且放入dbExpress组件以访问MS SQL Server

最后实现dbxUpProvider对象，程序代码如下：

```
unit udbxUpProvider;
{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ActiveX, Mtsobj, Mtx, ComObj, pdbxUpdateObjects_TLB, StdVcl;

type
  TdbxUpProvider = class(SMtsAutoObject, IdbxUpProvider
  protected
    procedure UpdateEmployee (vData: OleVariant; var sError: WideString;
      safecall;
  end;
```


implementation

```
uses ComServ, udmdbxUpProvider, DbClient, DB, Variants, SysUtils;
```

const

```
    sUpdateEmployee = 'update employee set ';
```

```
procedure TdbxUpProvider.UpdateEmployee (vData: OleVariant;
```

```
    var sError: WideString
```

var

```
    aCDS : TClientDataSet;
```

```
function GetUpdatesQL : String;
```

var

```
    iCount : Integer;
```

```
    vValue : Variant;
```

```
    aField : TField;
```

```
function GetFieldValue : String;
```

begin

```
    case aField.DataTypeof
```

```
        ftString, ftWideString :
```

```
            Result := ' = ' + ''' + VarToStr(vValue) + ''' + ',';
```

```
        ftSmallint, ftInteger :
```

```
            Result := ' = ' + IntToStr(vValue) + ',';
```

```
        ftFloat :
```

```
            Result := ' = ' + FloatToStr(vValue) + ',';
```

```
        ftTimeStamp :
```

```
            Result := ' = ' + ''' + DateTimeToStr(vValue) + ''' + ',';
```

```
    end;
```

end;

begin

```
    Result := sUpdateEmployee;
```

```
    for iCount := 0 to aCDS.FieldCount - do
```

begin

```
        aCDS.Next;
```

```
        if (VarIsNull (aCDS.Fields[iCount].Value)) then
```

begin

```
            aCDS.Prior;
```

```
            vValue := aCDS.Fields[iCount].Value;
```

end

else

begin

```

        vValue := aCDS.Fields[iCount].Value;
        aCDS.Prior;
    end;

    aField := aCDS.Fields[iCount];
    Result := Result + aField.FieldName + GetFieldValue;
end;

Delete (Result, Length(Result), 1);
aField := aCDS.FieldByName('emp_id');
vValue := aField.Value;
Result := Result + ' where emp_id ' + GetFieldValue;
Delete (Result, Length(Result), 1);
end;

begin
    try
        dmdbxUpProvider := TdmdbxUpProvider.Create(nil);
        aCDS := TClientDataSet.Create(nil);
        aCDS.Data := vData;
        try
            while not aCDS.Eof do
                begin
                    dmdbxUpProvider.sqldsGeneral.CommandText := GetUpdateSQL;
                    sError := dmdbxUpProvider.sqldsGeneral.CommandText;
                    dmdbxUpProvider.sqldsGeneral.ExecSQL (True);
                    aCDS.Next;
                    aCDS.Next;
                end;
            finally
                FreeAndNil (aCDS);
                FreeAndNil (dmdbxUpProvider);
            end;
            SetComplete;
        except
            on e : Exception do
                begin
                    sError := e.Message;
                    SetAbort;
                end;
            end;
        end;
    end;

initialization

```

```
TAutoObjectFactory.Create (ComServer, TdbxUpProvider, Class_dbxUpProvider,  
    ciMultiInstance, tmBoth  
end.
```

TdbxUpProvider的UpdateEmployee方法首先建立项目中的数据模块，再建立一个暂时的TClientDataSet组件以存储传递来的客户端修改数据信息 Delta，接着调用GetUpdateSQL内嵌方法以便根据Delta来组成正确的Update SQL语句，最后将SQL语句指定给sqlldsGeneral组件并且调用它的ExecSQL方法要求MS SQL Server执行更新数据的SQL语句。最后有两行aCDS.Next程序代码，这是因为Delta中对于每一次修改的数据保存了2个记录，因此当UpdateEmployee处理完一次修改的数据之后，必须在Delta中跳过2个记录才能到达下一个需要更新的记录，这在前面章节中介绍DataSnap原理时已经说明过了。

GetUpdateSQL则稍微复杂一点，它的工作是根据Delta中的数据形成正确的Update SQL语句。因此，GetUpdateSQL必须同时根据Delta中的原始数据以及修改后的数据来组成SQL语句，此外GetUpdateSQL还必须判断字段的类型才能指定字段值。在上面的程序代码中只处理了字符串类型、整数类型、浮点类型以及TimeStamp类型的字段，读者可以自行加入处理其他类型的字段的程序代码。当然，在这里本范例展示的是如何使用程序代码的方式来组成SQL语句。如果读者不想这么麻烦，那么可以把sqlldsGeneral连接到employee数据表，让sqlldsGeneral先取得字段的元数据信息，如此一来就不需要使用Object Pascal程序代码来自行判断字段类型了。

完成了上面的实现之后，同样编译并且注册这个新的COM+对象。

那么再回到客户端应用程序项目，并且修改客户端应用程序，加入如图5-21所示的控件准备通过TdbxProvider对象调用TdbxUpProvider对象来更新employee的数据。

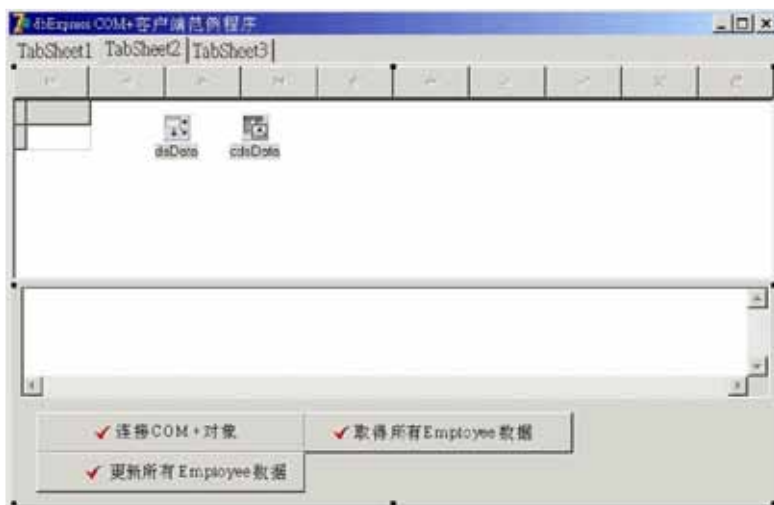


图5-21 修改范例客户端应用程序以执行更新 employee数据表的功能

下面列出 TdbxProvider 的 updateEmployee 方法，TdbxProvider 在被客户端调用之后会先建立 TdbxUpProvider 对象，然后调用 TdbxUpProvider 对象的 UpdateEmployee 方法来更新数据，并且把客户端传递来的 Delta 传递给 UpdateEmployee 方法。

```
procedure TdbxProvider.updateEmployee (vData: OleVariant;  
var  
    aUpObj : TdbxUpProvider;  
    sError : WideString;  
begin  
    try  
        OleCheck (ObjectContext.CreateInstance (CLASS_dbxUpProviderID, TdbxUpProvider, aUpObj));  
        aUpObj.UpdateEmployee (vData, sError);  
        SetComplete;  
    except  
        SetAbort;  
    end;  
end;
```

现在再次执行客户端应用程序，先连接 COM+ 对象，再取得 employee 数据，修改其中的任何数据，最后再点击“更新 Employee 数据”按钮，那么就可以成功地把客户端修改的数据更新回 MS SQL Server 了（见图 5 22）。



图5-22 执行范例客户端应用程序并且更新 employee 数据表中的数据

现在读者应该已经了解了如何在 COM+ 中使用 dbExpress。由于本书不是专门讨

论如何使用 Delphi 开发 COM+ 应用系统的书籍，因此读者如果有兴趣进一步了解 COM+ 技术，那么可以参考其他专门的 COM+ 书籍。不过在这里笔者也要提醒读者一些在 COM+ 中使用 dbExpress 时应该要注意的事情，这些事项如下：

- dbExpress For MS SQL Server 目前只支持 Local Transaction Service 接口，尚未完全支持分布式 Transaction Service 接口，因此在 COM+ 中无法加入分布式事务的功能。这代表目前的 dbExpress For MS SQL Server 可能无法依靠 COM+ 的事务功能在 COM+ 中正常地 Commit 或是 Rollback 数据。笔者已向 Borland 的 dbExpress 开发小组反映了目前 dbExpress For MS SQL Server 的这个限制，在未来的 Delphi Patch 中 Borland 可能会修正这个问题。
- 因此现在在 COM+ 中 dbExpress 必须使用自己的事务功能来 Commit 或是 Rollback 数据。

5.5 结论

本章中讨论了许多重要的概念和技术，包括 dbExpress 的事务功能，如何处理 dbExpress 的错误状态，以及如何在 COM+ 中使用 dbExpress 等。

对于事务方面，由于每一个关系型数据库提供的能力有所不同，因此读者在使用 dbExpress 提供的事务功能时必须注意使用的后端数据源是否提供了特定的事务能力。例如，dbExpress 可以提供嵌套事务功能，但不是每一个数据源都可以支持嵌套事务。另外，dbExpress 的 TransIsolation 功能也对事务影响很大，读者也必须了解 TransIsolation 对数据访问的影响。

在使用 dbExpress 开发数据库应用系统时，不可避免地都可能会遇到发生错误的情况，例如数据表中的数据已经被其他用户删除了，或是特定的字段值被改变了，导致数据无法修改回数据源。因此程序员在使用 dbExpress 时必须知道如何处理这些错误状况，以保证数据的完整性以及向用户提供更为稳定的应用程序。Delphi/Kylix 和 dbExpress 已经提供了许多方便的机制允许程序员妥善处理 dbExpress 的错误情形。

最后，本章讨论了如何在 COM+ 中使用 dbExpress。dbExpress 的性能搭配 COM+ 的对象缓冲池（Object Pooling）、线程缓冲池（Thread Pooling）以及连接缓冲池（Connection Pooling）实在是很完美的组合，可以提供非常有效率的 COM+ 应用系统。不过目前 dbExpress 只支持 COM+ Local Transaction 接口，这是读者在 COM+ 中使用 dbExpress 时要注意和了解的地方。

第6章 使用dbExpress处理复杂的数据类型

前面讨论处理数据的章节中，大都只是说明如何处理单档的数据（即单一数据表（Single Table）数据），或是一些简单的数据处理工作。但是在许多数据库应用系统中，程序员必须处理复杂的数据类型，例如主从应用，或是需要同时处理多重数据表的数据。dbExpress在处理复杂的数据类型时，有许多需要程序员特别注意的地方，同时也蕴含了程序员必须了解和熟悉的许多技巧。

本章讨论的内容就在于让读者了解如何使用 dbExpress处理复杂的数据，在本章中读者将会学习 TDataSetProvider组件对于 DataSnap的影响、主从数据类型以及 dbExpress和DataSnap如何处理多重数据表的问题。在了解了 DataSnap/dbExpress处理复杂数据类型的技术之后，读者应该可以使用 DataSnap/dbExpress处理任何类型的数据库应用系统了。

6.1 TDataSetProvider组件

在前面的章节中，本书大都是使用 TSimpleDataSet来展示如何处理数据，Delphi 6的用户则可以使用 TSQLClientDataSet组件来处理数据。但是在比较复杂的应用中，使用 TClientDataSet和TDataSetProvider组件能够帮助程序员解决一些比较困难的问题。在Delphi 7的TSimpleDataSet组件中虽然也提供了一些 TDataSetProvider组件的功能，但是在高级的应用中毕竟不如直接使用 TDataSetProvider/TClientDataSet方便。更何况 TSimpleDataSet提供的许多功能的概念也来自 TDataSetProvider组件提供的功能。因此了解如何善用 TDataSetProvider组件不但能够让读者更了解如何配合使用 TClientDataSet和TDataSetProvider组件，也能够帮助读者更好地掌握 TSimpleDataSet，因此本小节讨论的内容就在于让读者切实掌握 TDataSetProvider组件，也就是 dbExpress中扮演“数据提供者”的组件。

TDataSetProvider组件提供了许多功能，它主要的目的是向客户端 TClientDataSet组件提供数据以便进行处理数据的工作。下面列出了 TDataSetProvider组件最主要的功能：

- 向客户端 TClientDataSet组件提供从数据源取得的数据。
- 提供数据事件处理函数让 TClientDataSet控制如何访问数据以及如何将修改的数据更新回数据源。

- 根据TClientDataSet组件的Delta内容自动产生正确的SQL语句，让dbExpress执行SQL语句以便把客户端修改的数据更新回数据源。

由于TDataSetProvider组件向TClientDataSet提供了控制如何访问数据的能力，因此TDataSetProvider组件允许客户端的TClientDataSet微调处理数据的行为，这使客户端能够更细致地处理数据，也会影响性能，在稍后讨论TDataSetProvider相关的特性时会进一步说明。

此外，TDataSetProvider最重要的功能之一就是可以根据TClientDataSet传递的Delta内容自动产生SQL语句，让dbExpress驱动程序执行它以完成修改数据的工作。由于dbExpress引擎本身并不像BDE/IDAPI一样会根据修改的数据自动产生SQL语句，因此dbExpress需要TDataSetProvider的帮助来产生SQL语句，否则就需要客户端的TSimpleDataSet/TClientDataSet直接下达dbExpress要执行的SQL语句。但是在一般的应用中，客户端应用程序大都是直接调用TSimpleDataSet/TClientDataSet的ApplyUpdates方法来更新数据，因此就需要TDataSetProvider的帮助来产生正确的SQL语句。

从上面的描述读者可以了解到TDataSetProvider组件的重要性。虽然在一般的应用中程序员可能不会使用TDataSetProvider控制对数据的访问，但是在许多高级的应用中TDataSetProvider组件却有非常大的用处。如果读者能够了解TDataSetProvider提供的事件处理函数和特性，以及TDataSetProvider处理数据的流程，那么就可以充分地掌握TDataSetProvider组件，并且使用它来进行深入的应用。在下面的小节中将介绍TDataSetProvider组件中重要的事件处理函数和特性。

6 1 1 TDataSetProvider的重要事件处理函数

由于TDataSetProvider组件的重要功能之一是向TClientDataSet组件提供数据，因此TDataSetProvider当然提供了许多事件处理函数允许TClientDataSet使用各种方式取得数据。在一般的应用中TClientDataSet只需要设置ProviderName特性值为特定TDataSetProvider的名称，TClientDataSet就可以通过这个指定的TDataSetProvider从数据源取得数据。但是程序员也可以直接通过TDataSetProvider的事件处理函数来取得客户端需要的特定数据，这使TClientDataSet拥有了更细致的数据访问能力。

1. GetRecords事件处理函数

当TClientDataSet和TDataSetProvider组件合作使用DataSnap和dbExpress从数据源访问数据时，TClientDataSet和TDataSetProvider分别会触发数个事件处理函数让程序员能够控制或是微调访问数据的行为。在TClientDataSet向TDataSetProvider访问数据的过程中，TClientDataSet和TDataSetProvider会分别触发BeforeGetRecords以及AfterGetRecords事件处理函数，如图6 1所示。

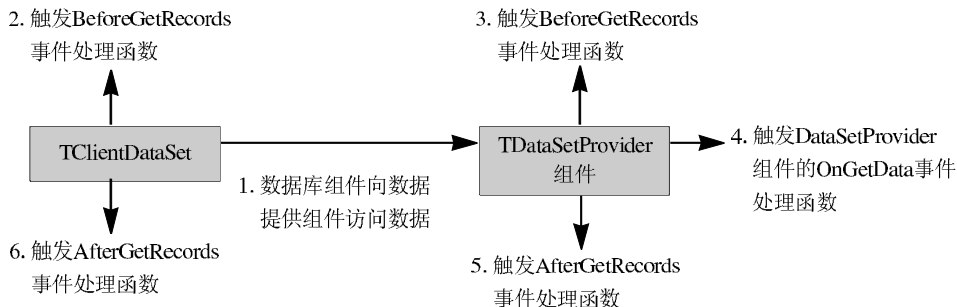


图6-1 TDataSetProvider组件访问数据时触发的事件处理函数

BeforeGetRecords以及AfterGetRecords事件处理函数声明为如下的原型：

```

procedure TdmdspDemo.cdsPerftestBeforeGetRecords (Sender: TObject;
  var OwnerData: OleVariant);
procedure TdmdspDemo.cdsPerftestAfterGetRecords (Sender: TObject;
  var OwnerData: OleVariant);
  
```

这两个事件处理函数都接受一个 OwnerData 参数，OwnerData 允许 TClientDataSet 传递特定的信息给 TDataSetProvider，也允许 TDataSetProvider 返回特定的信息给 TClientDataSet。程序员可以使用 OwnerData 来传递应用程序特有的信息，以控制 TDataSetProvider 如何访问数据源中的数据。

2. DataRequest 事件处理函数

除了 BeforeGetRecords 以及 AfterGetRecords 事件处理函数之外，TClientDataSet 的 DataRequest 方法以及 TDataSetProvider 的 OnDataRequest 事件处理函数也允许客户端应用程序使用定制的方式来访问数据。在一般的应用中，当 TDataSetProvider 组件连接了 TSQLDataSet 或是 TSQLQuery 之后，通常在 TSQLDataSet 或是 TSQLQuery 中使用固定的 SQL 语句来访问数据。而 TDataSetProvider 的 OnDataRequest 却允许程序员使用动态的方式来处理访问数据的动作。图 6 2 说明了 TClientDataSet 的 DataRequest 方法以及 TDataSetProvider 的 OnDataRequest 事件处理函数的工作流程。

TClientDataSet 的 DataRequest 方法声明为如下的原型：

```

function DataRequest (Data: OleVariant): OleVariant virtual;
  
```

其中 OleVariant 类型的参数 Data 是 TClientDataSet 可以传递给 TDataSetProvider 的任何信息，例如 TClientDataSet 可以传递给 TDataSetProvider 动态 SQL 语句或任何键值信息。

TDataSetProvider 的 OnDataRequest 事件处理函数则有如下的声明：

```

function TdmdspDemo.dspPerftestDataRequest (Sender: TObject;
  Input: OleVariant): OleVariant;
  
```

在 TDataSetProvider 的 OnDataRequest 事件处理函数中，Input 参数就是由 TClientDataSet 传递来的信息，TDataSetProvider 组件可以使用这个参数值来重设它

连接的TSQLDataSet/TSQLQuery使用的SQL语句，或是使用TClientDataSet传递来的键值信息搜寻数据。

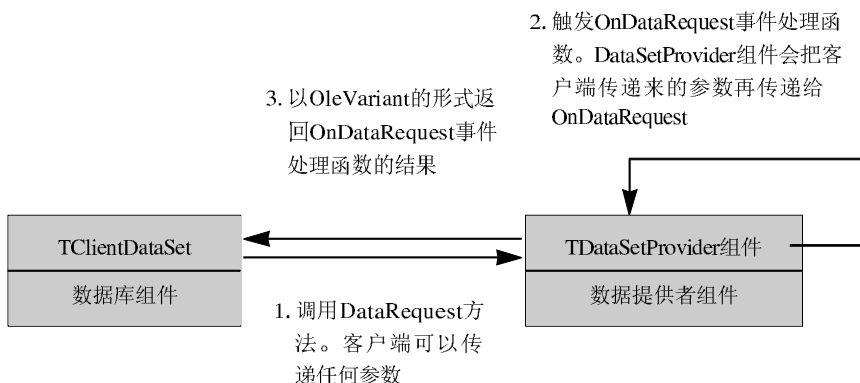


图6-2 TDataSetProvider组件提供定制访问数据的能力

程序员也可以直接使用TClientDataSet的CommandText以及TDataSetProvider的Options\poAllowCommandText特性值来达到相同的目的，因此现在较少使用OnDataRequest。但是由于DataSnap和dbExpress也可以在分布式应用系统中使用，例如COM+或是DataSnap Server中，而OnDataRequest在这类应用程序中拥有较大的弹性以及可重复使用的特点。

3. DataSnap处理数据修改的流程

当程序员使用TClientDataSet调用ApplyUpdates方法把客户端的数据更新回数据源时，事实上DataSnap会触发TClientDataSet和TDataSetProvider组件中数个重要的事件处理函数，让程序员能够控制修改的数据如何更新回数据源中。如果程序员没有使用TClientDataSet和TDataSetProvider的任何事件处理函数来控制或是微调更新流程，那么DataSnap便会使用默认的方式来更新数据。

简单地说，如果程序员使用DataSnap默认的方法将数据更新回数据源，DataSnap会根据数据源的元数据信息以及修改的数据来自动产生SQL语句，再使用这些产生的SQL语句把数据更新回数据源中。不过在DataSnap的整个更新流程中，DataSnap会触发数个事件处理函数让程序员决定是否使用DataSnap产生的SQL语句，或是让程序员自己决定如何执行更新数据的动作，这允许程序员使用更有效率的方法更新数据，或是在更新数据的过程中让程序员有机会进行额外的处理工作，例如检查数据或是对数据进行加密/解密等。另外的好处是这些DataSnap事件处理函数允许程序员处理DataSnap无法处理的复杂数据类型，例如Join的数据或是更为复杂的应用等。在稍后的章节中，读者会了解如何利用这个特点来解决这些问题以及如何增加DataSnap的性能。

图6 3详细描述了TDataSetProvider和TDataSetProvider修改数据的流程，从图6 3

中读者可以看到, 当 `TDataSetProvider` 更新数据时会触发 `TDataSetProvider` 的 `BeforeUpdateRecord` 事件处理函数, 以便让程序员有机会再次处理即将更新的数据, 或是使用程序员自己的程序代码来更新数据, 而不由 `DataSnap` 来更新数据。对于每一个要更新的记录 (即存在于 `TClientDataSet` 的 `Delta` 特性值中的修改数据), `TDataSetProvider` 都会触发一次 `BeforeUpdateRecord` 事件处理函数。不过要想让 `TDataSetProvider` 触发 `BeforeUpdateRecord` 事件处理函数以允许程序员有机会自行处理修改的数据, 程序员必须先设置 `TDataSetProvider` 组件的 `ResolveToDataSet` 特性值为 `True`, 而 `ResolveToDataSet` 特性值的默认设置是 `False`。

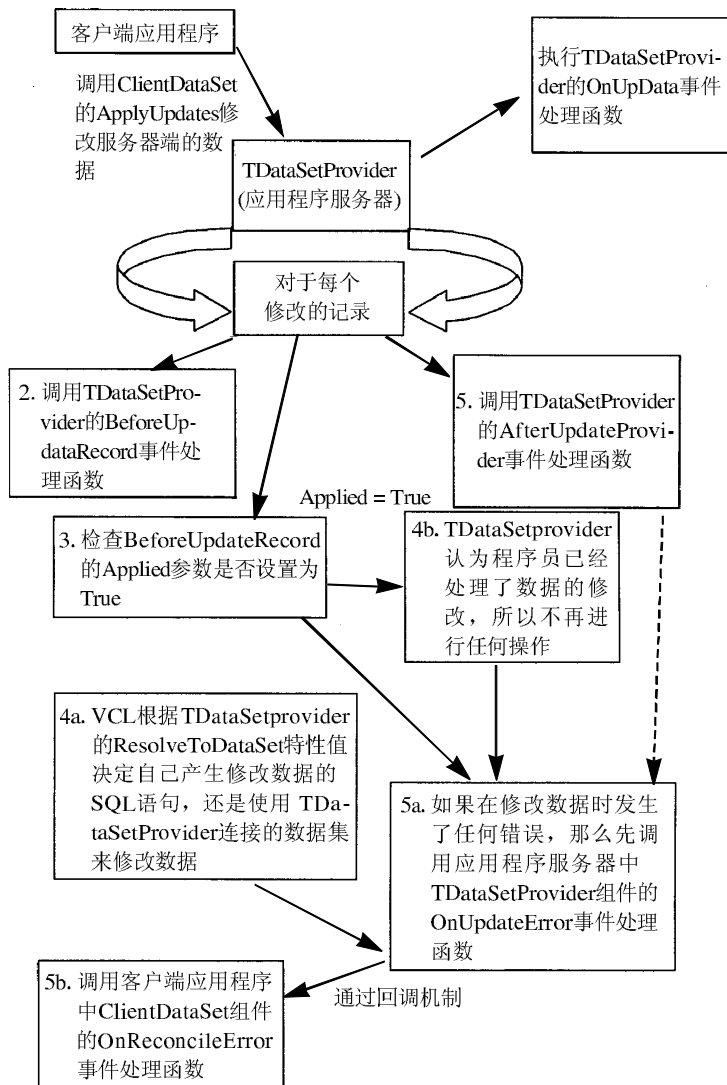


图6-3 TDataSetProvider组件将数据更新回数据源的流程

在BeforeUpdateRecord事件处理函数中, 在执行完程序员的程序代码之后, 程序员仍然可以决定再让 DataSnap进行真正更新数据的工作, 而不是由程序员自己来更新。要想如此做, 程序员可以使用 BeforeUpdateRecord事件处理函数中的特定参数。在说明之前, 先让我们看看 BeforeUpdateRecord事件处理函数的声明原型:

```
type TBeforeUpdateRecordEvent = procedure (Sender: TObject; SourceDS: TDataSet;  
DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean of  
object;
```

在上面的BeforeUpdateRecord事件处理函数中, 参数 Applied就可以决定是由程序员完成更新数据的工作, 还是再让 DataSnap更新数据。如果程序员只是在更新数据之前再执行一些额外的工作, 那么在 BeforeUpdateRecord事件处理函数中执行完程序员的程序代码之后, 程序员可以设置 Applied参数为False以告诉DataSnap程序员没有自己更新数据, 仍然需要由 DataSnap进行更新数据的工作。如果程序员已经在 BeforeUpdateRecord事件处理函数中更新了数据, 那么就必须设置 Applied为True, 以避免DataSnap再次执行更新数据的动作。

BeforeUpdateRecord事件处理函数中的 SourceDS代表数据源中的原始数据, DeltaDS代表客户端修改的数据, 而 UpdateKind参数则代表目前修改数据的种类, 例如是新增的数据、修改的数据或是被删除的数据。

当TDataSetProvider组件自动产生SQL语句更新数据时, TDataSetProvider组件也会参考数个特性值来决定如何产生 SQL语句, 其中最重要的就是 TField对象的 ProviderFlags特性值以及TDataSetProvider自己的UpdateMode特性值。

当TDataSetProvider自动产生SQL语句时, 会在SQL语句中产生动态参数以便使用Delta中用户修改的数据填入正确的数值。由于 Delta中的数值包含了未修改的数据以及修改过或是新增的数据, 因此 TDataSetProvider必须能够知道将什么数值填入每一个动态参数中。而 TField的ProviderFlags特性值就可以决定当 TDataSetProvider产生SQL语句并且将 Delta中的修改数据代入 SQL语句的动态参数时要填入什么数值。图6 4说明了TDataSetProvider组件在产生SQL语句时的流程。

在下面的小节中将说明 TField的ProviderFlags以及TDataSetProvider的UpdateMode如何影响TDataSetProvider产生的SQL语句。

6 1 2 TDataSetProvider的重要特性

上一小节说明了 TDataSetProvider组件处理数据时的流程以及触发的事件处理函数, 本小节中将讨论两个会影响 TDataSetProvider组件如何产生更新数据的 SQL语句的重要特性。

1. TDataSetProvider的UpdateMode特性值

TDataSetProvider的UpdateMode特性值控制当 TDataSetProvider产生SQL语句时

在SQL语句的where子句中产生的过滤条件。下面的表格列出了 TDataSetProvider的 UpdateMode特性可以设置的值以及它们的意义。

UpdateMode值	说 明
UpWhereAll	所有旧字段的数值都必须符合
upWhereChanged	只有键值字段和被修改的字段值必须符合
upWhereKeyOnly	只有键值字段的值必须符合

UpdateMode的默认值是upWhereAll，现在让我们解释 UpdateMode特性值的意思。

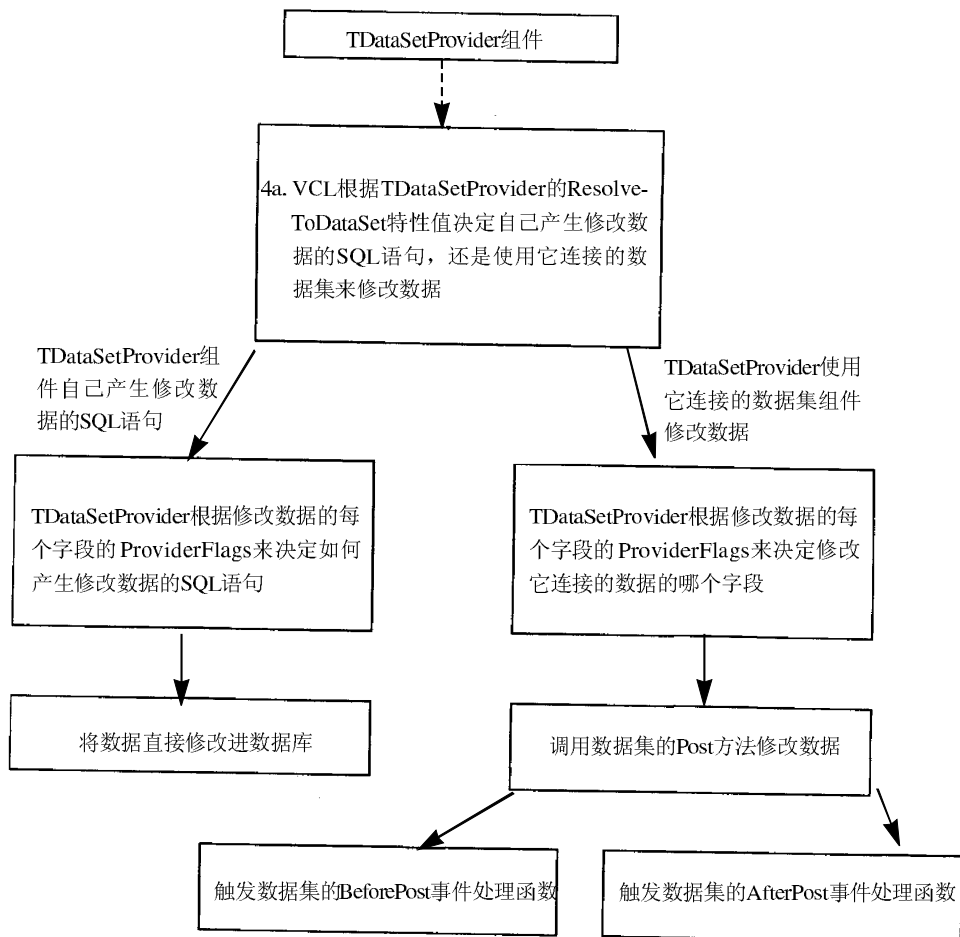


图6-4 TDataSetProvider组件产生SQL语句的流程

当TDataSetProvider组件在产生SQL语句以便把修改的数据更新回数据源时，DataSnap会试着先在数据源中搜寻这个要修改的记录的原始数据（当然，这是对要修改和删除的数据而言，新增的数据则无需进行这个动作），而UpdateMode就是控制DataSnap如何在数据源中搜寻原始的数据。让我们使用一个范例来说明，假设下

面的表格是数据源中的 Employee 数据表的数据，在其中假设 Name 字段是索引字段。

Name	Age	Occupation
李维	34	资深技术顾问
李匡正	30	资深产品经理

假设现在客户端的用户使用 dbExpress 访问 Employee 数据表，并且修改了其中李维的 Occupation 字段的数据，那么当客户端调用 TSimpleDataSet/TClientDataSet 的 ApplyUpdates 方法后，Delta 中便会包含下面的内容。

Name	Age	Occupation
李维	34	资深技术顾问/PLSM
		资深技术顾问

当 TDataSetProvider 自动产生 SQL 语句时，UpdateMode 不同的设置值便会产生如下的结果。

UpdateMode 值	产生的 where 句子
upWhereAll	Where Name= “李维” and Age = 34 and Occupation = “资深技术顾问”
upWhereChanged	Where Name= “李维” and Occupation = “资深技术顾问”
upWhereKeyOnly	Where Name= “李维”

由于在数据库应用系统中可能同时有许多人都在修改数据源中的数据，因此 UpdateMode 特性值便会影响数据是否能够成功地更新回数据源中。例如在上面的更新过程中，如果同时有另外一个用户 B 将李维的 Age 字段更新为 35，如果 UpdateMode 特性值设置为 upWhereAll，那么 upWhereAll 会使用 Age = 34 来搜寻原始的数据，但是这将无法找到，因为用户 B 已经修改了这个字段的值，因此 DataSnap/dbExpress 会显示“数据已经被其他用户修改”的错误。但是如果 UpdateMode 设置为 upWhereChanged 或是 upWhereKeyOnly，那么上述的数据就可以成功地更新回数据源中。

因此，UpdateMode 特性值是控制 TDataSetProvider 搜寻数据的严谨程度的机制。在默认情况下 TDataSetProvider 使用最严谨的设置。程序员在实际的应用中应该根据应用系统的需求来适当地设置 UpdateMode 的值，在有些情形中使用过于严谨的设置反而会造成应用系统的执行发生问题。

2. TField 的 ProviderFlags 特性值

TField 的 ProviderFlags 特性值用来控制和指定 TSimpleDataSet/TClientDataSet 中每一个字段的行为特性，根据这个行为特性 TDataSetProvider 组件便能够决定在设置动态参数时需要代入什么字段值。下面的表格列出了 ProviderFlags 特性值能够设置的值以及意义。

ProviderFlags值	说 明
pfInUpdate	这个字段是需要修改数据的字段
pfInWhere	这个字段是在寻找记录时使用的
pfInKey	这个字段属于键值字段
pfHidden	这个字段是隐藏的字段，一般的 TSimpleDataSet/TClientDataSet无法看到或是修改此字段的值

让我们再使用上面的范例来说明 **ProviderFlags**特性值的意义。当客户端的 **TSimpleDataSet/TClientDataSet**从数据源取得了 **Employee**数据表的数据之后，在 **TSimpleDataSet/TClientDataSet**的**TField**对象内部对于每一个字段便设置了如下的 **ProviderFlags**值。

字 段	ProviderFlags值
Name	[pfInUpdate, pfInKey, pfInWhere]
Age	[pfInUpdate, pfInWhere]
Occupation	[pfInUpdate, pfInWhere]

由于**Name**是索引字段，因此它的 **ProviderFlags**特性值包含了 **pfInKey**，而每一个字段都包含**pfInUpdate**代表这些字段都可以被修改，此外每一个字段也包含**pfInWhere**，这代表这些字段都可能会出现在 **TDataSetProvider**自动产生的SQL语句的**where**句子中。如果**TDataSetProvider**的**UpdateMode**特性值是**upWhereAll**的话，那么这些字段都会出现在**Where**句子中。

假设现在**Employee**是**Oracle**的数据表，那么由于**Oracle**的数据表中包含了代表每一个记录的唯一**RAWID**值，那么此时客户端的 **TSimpleDataSet/TClientDataSet**的**TField**对象中便会有如下的 **ProviderFlags**特性值。

字 段	说 明
RAWID	[pfInKey, pfHidden, pfInWhere]
Name	[pfInUpdate, pfInKey, pfInWhere]
Age	[pfInUpdate, pfInWhere]
Occupation	[pfInUpdate, pfInWhere]

由于客户端无法看到或是修改 **RAWID**，因此**RAWID**的**ProviderFlags**特性值必须设置为**pfHidden**。

一般来说，程序员并不需要修改 **ProviderFlags**的值，交由**DataSnap**来处理即可，不过程序员在需要时仍然可以使用程序代码来指定 **TField**对象的**ProviderFlags**值。例如，如果我们不想让上面的 **Name**字段可以被修改，那么就可以使用下面的程序代码设置**Name**字段的**ProviderFlags**值：

```
aCDS.FieldByName('Name').ProviderFlags := [pfInKey, pfInWhere];
```

6 1 3 TDataSetProvider的范例

在本小节中将使用一个简单的范例来展示前面讨论的一些内容，这个范例将显示 TField对象的 ProviderFlags值以及 UpdateMode特性值对于 TDataSetProvider产生的影响。图6 5是此范例的主窗体。在范例主窗体中将会显示 D7Books数据库中BOOKS数据表的 ISBN和BOOKNAME这两个字段的 ProviderFlags特性值，而下方的三个 TRadioButton控件则可以让用户选择使用什么 UpdateMode值。当用户修改了数据并且点击下方的 ApplyUpdates按钮之后，中间的 TMemo控件便会使用 TSQLMonitor来显示 TDataSetProvider产生的更新数据的 SQL语句。

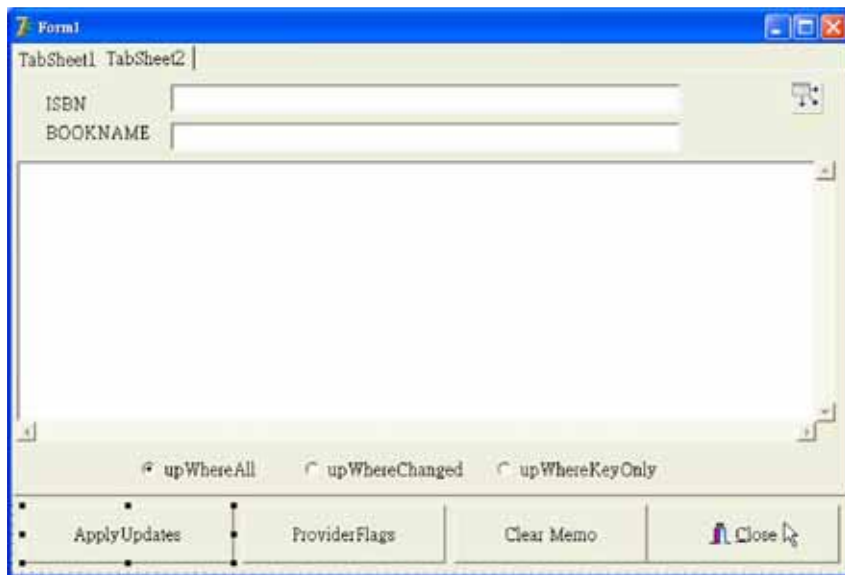


图6-5 范例程序的主窗体

下面是图6 5中“ProviderFlags”按钮的 OnClick事件处理函数的实现程序代码，在其中分别调用了 ShowISBNProviderFlags以及 ShowBookNameProviderFlags，以字符串的形式显示 ISBN和BOOKNAME这两个字段的 ProviderFlags特性值。

```
procedure TForm1.bbtnGetProviderFlagsClick(Sender: TObject);
begin
    ShowISBNProviderFlags;
    ShowBookNameProviderFlags;
end;

procedure TForm1.ShowBookNameProviderFlags;
var
    aField : TField;
begin
```

```

aField := dmdspDemo.cdsBooks.FieldByName('BOOKNAME');
edtBookName.Text := GetProviderFlagsStringFormat(aField.ProviderFlags);
end;

procedure TForm1.ShowISBNProviderFlags;
var
    aField : TField;
begin
    aField := dmdspDemo.cdsBooks.FieldByName('ISBN');
    edtISBN.Text :=
        GetProviderFlagsStringFormat (aField.ProviderFlags);
end;

function TForm1.GetProviderFlagsStringFormat (pf: TProviderFlags)
String;
begin
    Result := '[';
    if (pfInUpdate in pf) then
        Result := Result + 'pfInUpdate,';
    if (pfInWhere in pf) then
        Result := Result + 'pfInWhere,';
    if (pfInKey in pf) then
        Result := Result + 'pfInKey,';
    if (pfHidden in pf) then
        Result := Result + 'pfHidden,';

    if (Result[Length (Result)] = ',') then
        Delete (Result, Length (Result), 1);
    Result := Result + ']';
end;

```

下面是图 6 5 中“ApplyUpdates”按钮的OnClick事件处理函数的实现程序代码。在调用数据模块中的 TClientDataSet的ApplyUpdates方法之前，此程序代码先根据用户在主窗体中选择的 TRadioButton来设置UpdateMode的值。

```

procedure TForm1.bbtnApplyClick (Sender: TObject);
begin
    dmdspDemo.dspBooks.UpdateMode := GetUpdateMode;
    dmdspDemo.cdsBooks.ApplyUpdates (0);
end;

```

图 6 6到图 6 8是执行此范例程序，点击“ProviderFlags”按钮并且使用不同的UpdateMode特性值执行的结果。读者从每一个画面中间的 TMemo控件中可以看到，不同的UpdateMode设置值果然控制了TDataSetProvider组件产生的Where子句中的条件，而且ISBN和BOOKNAME字段的ProviderFlags特性值也显示在主窗体中。由于

ISBN字段是主键字段，因此它的 **ProviderFlags**特性值是 [pfInUpdate,pfInWhere,pfInKey]，而BOOKNAME只是一般的字段，因此它的 **ProviderFlags**特性值是 [pfInUpdate,pfInWhere]，这些完全符合前面我们讨论的有关 **ProviderFlags**特性值的内容。

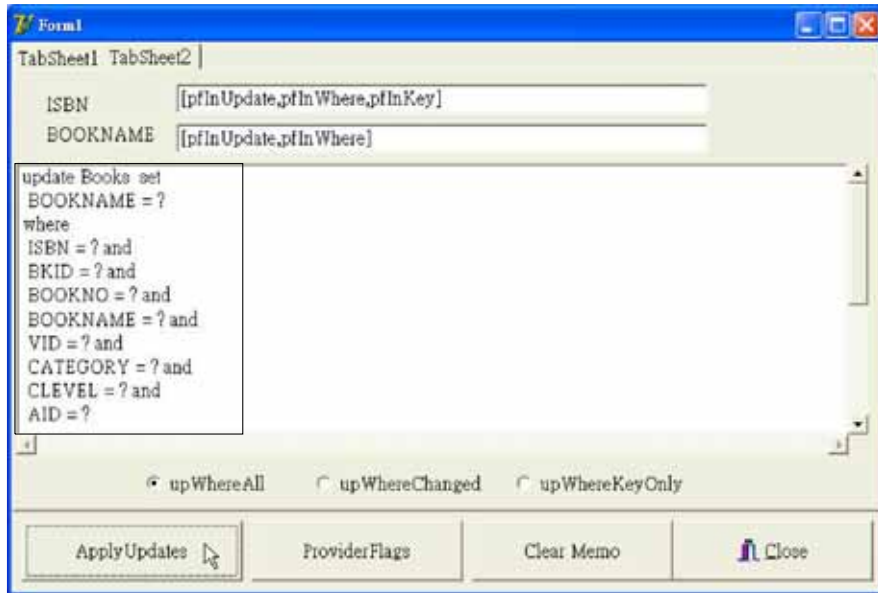


图6-6 UpdateMode使用upWhereAll值的结果

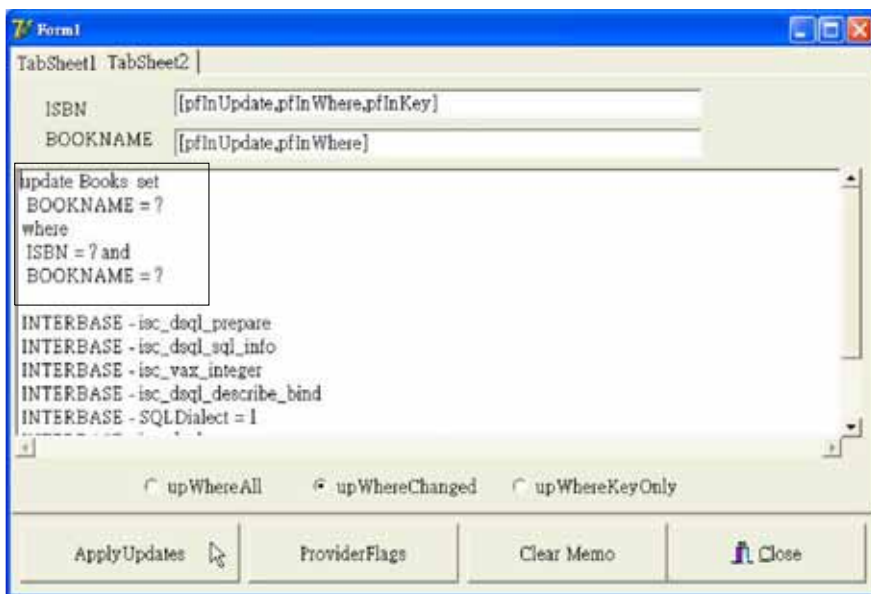


图6-7 UpdateMode使用upWhereChanged值的结果



图6-8 UpdateMode使用upWhereKeyOnly值的结果

6.2 主从类型的应用

主从（Master/Detail）类型的应用是许多数据库应用程序经常会使用的功能，其中最主要的应用是属于数据中一对多的需求，例如个人银行存提款信息、客户订单信息以及会计系统中的过帐数据等的应用。正是由于主从类型的应用是许多数据库应用程序经常使用的功能，因此 Delphi/Kylix 一直对这类的应用有很好的支持，从以往 Delphi 的 BDE/IDAPI 到 ADO 以及现在的 dbExpress，Delphi 一直提供了非常方便的功能来支持程序员实现主从类型的功能。

由于 Delphi/Kylix 已经在主从类型的应用方面提供了非常方便的支持，程序员基本上只需使用 Delphi/Kylix 提供的组件就可以直接在 dbExpress 应用程序中支持主从类型的功能，因此本小节将直接使用一个简单的范例来说明如何使用 dbExpress 组件来实现主从类型的功能。

6.2.1 使用组件和dbExpress实现主从功能

本范例要展示的是如何显示 D7Books 数据库中 PUBLISHERS 和 BOOKS 这两个数据表的数据。由于 PUBLISHERS 和 BOOKS 数据表中存在着主从数据的关系，这两个数据表可以通过外键字段 VID 来串连其中的数据，因此本小节将使用两个方法来实现主从功能，第一个范例完全使用 dbExpress 组件的功能来完成，程序员几乎不需要编写任何程序代码。第二个范例则使用了少许的程序代码来完成相同的功能。

首先在Delphi/Kylix中建立一个项目，再建立一个数据模块（见图69）。

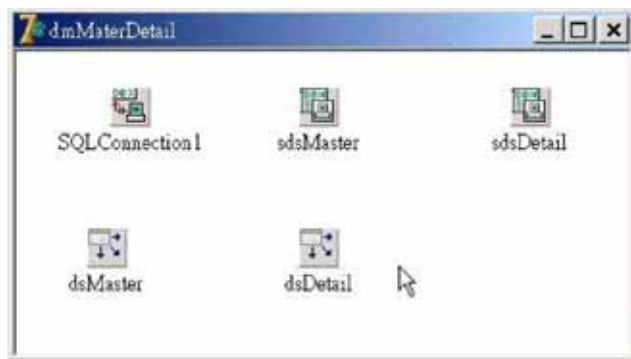


图6-9 在数据模块中放入dbExpress组件，设置TSimpleDataSet，
分别连到D7Books中的PUBLISHERS和BOOKS数据表

接着设置数据模块中的dbExpress组件的特性值，如下：

TSQLConnection:

特性名称	设置特性值
Name	SQLConnection1
ConnectionName	D7Books

TSimpleDataSet:

特性名称	设置特性值
Name	sdsMaster
Connection	SQLConnection1
DataSet\CommandText	select * from PUBLISHERS

TSimpleDataSet:

特性名称	设置特性值
Name	SdsDetail
Connection	SQLConnection1
DataSet\CommandText	select * from BOOKS

然后设置dsMaster的DataSet为sdsMaster，设置dsDetail的DataSet为sdsDetail。接下来，我们就必须为sdsMaster和sdsDetail设置主从关系，这可以通过TSimpleDataSet的MasterSource以及MasterFields来完成。

TSimpleDataSet的MasterSource特性值可以指定主数据的来源，因此我们必须先设置sdsDetail的MasterSource特性值为指向sdsMaster的TDataSource组件，那当然就是dsMaster了。而TSimpleDataSet的MasterFields特性值则可以定义主从数据表之间

的外键字段。由于PUBLISHERS和BOOKS数据表是以VID字段作为外键的。因此我们可以使用对象检视器点击sdsDetail的MasterFields特性，此时Delphi/Kylix会显示图6-10所示的对话框让程序员使用点击的方式定义数据表之间的外键。因此在图6-10所示的对话框中我们就从两个数据表中分别选择VID字段。

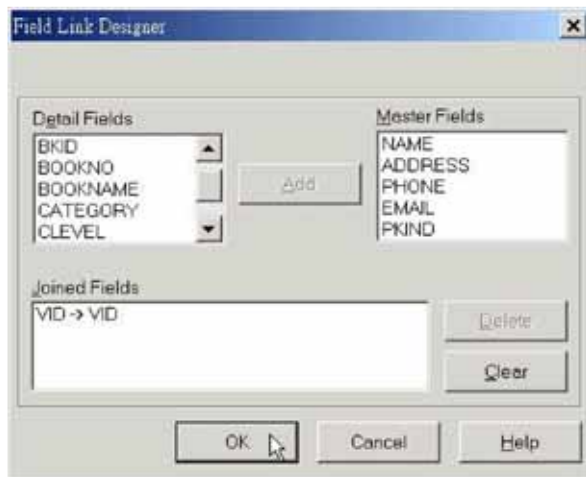


图6-10 设置sdsMaster和sdsDetail组件之间的MasterFields特性值

现在当我们点击数据模块中的sdsDetail组件时，可以看到图6-11所示的对象检视器，分别在MasterSource和MasterFields特性值中定义了正确的特性值。



图6-11 设置完成之后的sdsDetail组件

最后我们回到范例程序的主窗体，并且在其中放入如图6-12所示的控件。上方的TDBGrid控件连接的是sdsMaster，而下方的TDBGrid控件连接的是sdsDetail。

现在这个主从应用的数据库应用程序已经完成了，我们只需要编译并且执行程序就可以看到类似于图6-13的画面，当用户浏览上方的主数据时，下方的控件果然可以显示正确的从属数据。

除了完全使用dbExpress以及Delphi/Kylix的数据感知组件来完成主从应用之外，

在有的应用中为了设计和效率的因素程序员可能需要动手自行实现主从应用。这也很简单，因为这几乎只需要编写少量的程序代码。

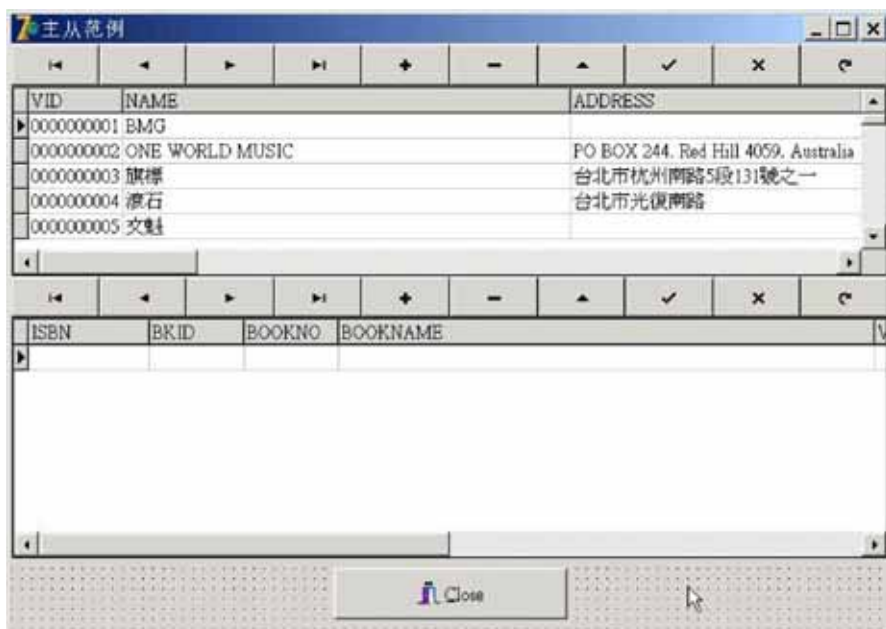


图6-12 范例程序的主窗体

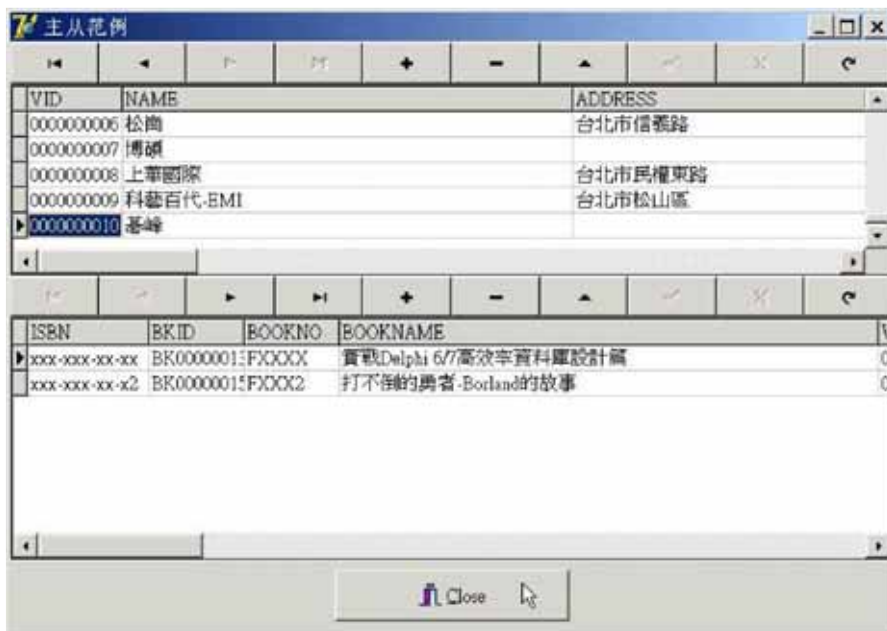


图6-13 执行范例的画面，显示主数据时果然可同时显示从属数据

6.2.2 使用程序代码实现主从功能

本小节将继续修改刚才的实现，并且使用少量的程序代码来完成主从应用。首先回到刚才的数据模块中，再放入另外一个 TSimpleDataSet，设置它的 SQLConnection 特性值为 SQLConnection1，再设置它的 Name 特性值为 sdsGeneral，如图 6-14 所示。

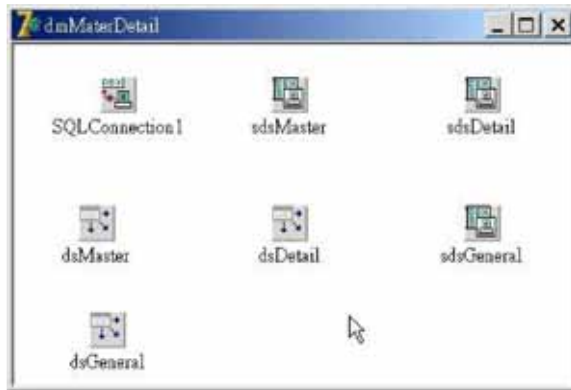


图6-14 在数据模块中放入另外一个 TSimpleDataSet

接着设置 sdsGeneral 的 DataSet\CommandText 特性值如下：

```
select * from Books where VID = :VID
```

sdsGeneral 组件将根据目前 sdsMaster 指向的 PUBLISHERS 记录到 BOOKS 数据表中搜寻从属数据。接着在数据模块中实现 sdsMaster 的 AfterOpen、AfterScroll 以及数据模块的 OnCreate 以及 OnDestroy 事件处理函数，程序代码如下：

```
procedure TdmMaterDetail.GetDetail;
begin
    sdsGeneral.Active := False;
    try
        sdsGeneral.DisableControls;
        try
            sdsGeneral.DataSet.Params.ParamByName('VID').AsString :=
                sdsMaster.FieldByName('VID').AsString;
            sdsGeneral.Active := True;
        except
            ;
        end;
    finally
        sdsGeneral.EnableControls;
    end;
end;

procedure TdmMaterDetail.sdsMasterAfterOpen (DataSet: TDataSet);
```

```

begin
    GetDetail;
end;

procedure TdmMaterDetail.sdsMasterAfterScroll (DataSet: TDataSet;
begin
    GetDetail;
end;

procedure TdmMaterDetail.DataModuleCreate (Sender: TObject;
begin
    sdsGeneral.DataSet.Prepared := True;
end;

procedure TdmMaterDetail.DataModuleDestroy (Sender: TObject;
begin
    sdsGeneral.Active := False;
    sdsGeneral.DataSet.Prepared := False;
end;

```

为了性能因素，在数据模块的 OnCreate 事件处理函数中我们首先准备 sdsGeneral 的 SQL 语句，这样可以有较好的性能，因为 sdsGeneral 中的 SQL 语句会随着用户浏览 PUBLISHERS 数据而不断地执行。当然在数据模块的 OnDestroy 事件处理函数中也必须设置 Prepared 特性值为 False 以释放后端数据源的资源。



图6-15 修改后范例程序的执行画面

当用户在主窗体中浏览数据时以及 `sdsMaster` 首次开启时，我们都必须根据 `sdsMaster` 中目前的记录到 `BOOKS` 数据表中搜寻相对应的数据，并且显示在主窗体的数据感知组件中。因此在 `sdsMaster` 的 `OnOpen` 以及 `OnAfterScroll` 事件处理函数中都调用了 `GetDetail` 方法，在 `GetDetail` 方法中简单地把目前 `PUBLISHERS` 记录的 `VID` 字段值代入 `sdsGeneral` 的动态参数中，再让数据源搜寻这个 `PUBLISHERS` 记录的从属数据。

现在再次编译并且执行这个范例，那么读者可以看到类似于图 6 15 的画面，修改过的范例程序果然和刚才一样提供了主从关系的功能。当然读者应该在刚才的 `GetDetail` 方法中调用 `sdsGeneral` 的 `DisableControls` 和 `EnableControls` 方法以避免数据感知组件无谓的更新动作，进一步增加范例程序的性能。

从上面讨论的内容中读者可以了解到，在 `dbExpress` 中开发主从类型的数据库应用程序是非常简单且方便的。

6.3 处理多数据表数据

现在读者应该已经知道 `dbExpress` 引擎是一个按照设计简单但是有效率的方式设计的，客户端丰富的数据处理能力是由 `DataSnap` 提供的。`dbExpress` 引擎只负责提供执行 `SQL` 语句和返回结果数据集的能力，因此对于一些复杂的数据处理工作无能为力。例如，客户端使用了 `SQL` 语句 `Join` 了数个数据表的数据，然后在客户端修改这些数据，再想要通过 `DataSnap` 的 `ApplyUpdates` 方法把这些 `Join` 过的数据更新回后端数据源，那么这是行不通的，因为 `DataSnap` 无法产生更新多个数据表的 `SQL` 语句。

因此在这种应用中，程序员如果想要把 `Join` 的数据更新回数据源的话，那么就必须自己处理更新多个数据表的工作。要完成这个工作，程序员可以把所有数据返回给数据源中的一个存储过程来更新数据，这样似乎比较有效率，但是这样做有几个问题。首先，数据源中的存储过程无法了解如何处理 `DataSnap` 的数据封包（`TSimpleDataSet/TClientDataSet` 的 `Data` 和 `Delta`）。第二个问题是，即使程序员能够让存储过程拆解 `DataSanp` 的数据封包，但是在存储过程中处理 `DataSanp` 的数据封包却不一定有效率，特别是当更新数据还需要进行数据的运算工作时。存储过程应该只用于处理大量的数据和进行少量计算，如果使用存储过程来处理一般的运算，那么是非常不明智的。

因此要处理 `Join` 过的数据，一个比较好的方法是在 `TDataSetProvider` 组件的事件处理函数中进行这种复杂的数据处理。在前面介绍 `TDataSetProvider` 的小节中已经说明了 `TDataSetProvider` 组件如何控制将客户端的修改数据更新回数据源的流程，由于 `TDataSetProvider` 组件的事件处理函数能够让程序员控制数据如何更新回数据源，因此我们就可以利用 `TDataSetProvider` 的特性来解决如何将 `Join` 数据更新回多个数据表

Set组件而使用TClientDataSet加上TDataSetProvider，那么请不要忘记设置TDataSetProvider的ResolveToDataSet特性值为True，这样TDataSetProvider才会触发BeforeUpdateRecord事件处理函数。

接着在主窗体中放入图6-17所示的控件，在TDBCtrlGrid中的各个控件大部分是来自EMP数据表，但是其中的DNAME和LOC控件则是来自DEPT数据表。此外在主窗体中也放入了两个TClientDataSet以及TDBGrid等控件，用来观察数据模块中sqlcdsJoins组件的Data以及Delta特性值在用户修改Join的数据时的改变情形。

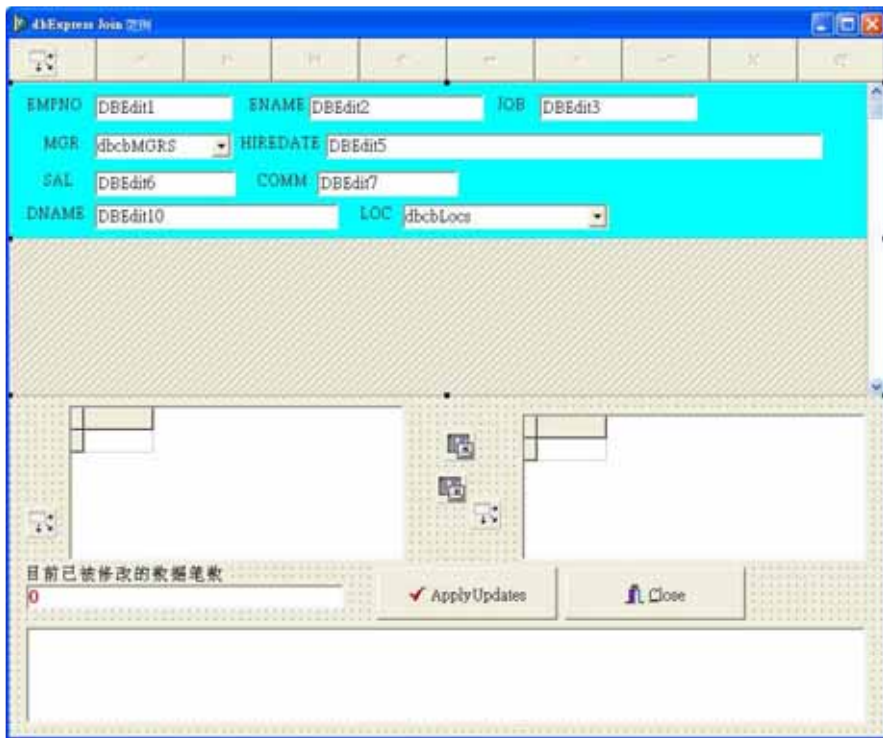


图6-17 范例程序的主窗体

下面是范例主窗体的实现程序代码。在下面的程序代码中，当范例程序执行时它首先从Oracle数据库的DEPT以及MANAGERS数据表中取出数据，分别显示在主窗体的MGR和LOC下拉框中让用户可以选择数据。其中最主要的部分应该是当用户在主窗体中修改了Join的数据之后，可以点击主窗体中的“ApplyUpdates”按钮将数据更新回数据源中。

“ApplyUpdates”按钮的事件处理函数调用数据模块中sqlcdsJoins组件的ApplyUpdates方法进行更新。由于sqlcdsJoins是使用select SQL语句从两个不同的数据表中选取数据，因此调用sqlcdsJoins的ApplyUpdates方法将无法由DataSnap自动产生SQL语句来更新Join数据，因此我们必须实现sqlcdsJoins组件的BeforeUpdate

Record事件处理函数。

```
procedureTfrmJoinMain.FillLocs;
begin
  dmJoinDemos.sqldsLocs.Active := True;
  try
    while not dmJoinDemos.sqldsLocs.Eof do
    begin
      Self.dbcbLocs.Items.Add (dmJoinDemos.sqldsLocs.Fields[0].AsString);
      dmJoinDemos.sqldsLocs.Next;
    end;
  finally
    dmJoinDemos.sqldsLocs.Active := False;
  end;
end;

procedureTfrmJoinMain.FillManagers;
begin
  dmJoinDemos.sqldsMGRS.Active := True;
  try
    while not dmJoinDemos.sqldsMGRS.Eof do
    begin
      Self.dbcbMGRS.Items.Add (dmJoinDemos.sqldsMGRS.Fields[0].AsString);
      dmJoinDemos.sqldsMGRS.Next;
    end;
  finally
    dmJoinDemos.sqldsMGRS.Active := False;
  end;
end;

procedureTfrmJoinMain.FormActivate (Sender: TObjcet;
begin
  FillManagers;
  FillLocs;
end;

procedureTfrmJoinMain.bbtnApplyClick (Sender: TObject);
begin
  if (dmJoinDemos.sqlcdsJoins.ChangeCount) > 0 then
  begin
    dmJoinDemos.sqlcdsJoins.ApplyUpdates (0);
  end;
end;
```

sqlcdsJoins的BeforeUpdateRecord事件处理函数是在范例程序的数据模块中实现的，下面是范例程序数据模块的完整程序代码：

```
unit udmJoinDemo;

interface

uses
  SysUtils, Classes, DBXpress, Provider, SqlExpr, DB, DBClient, DBLocal,
  DBLocalS, FMTBcd, Dialogs;

type
  TdmJoinDemos = class (TDataModule)
    sqlcJoins: TSQLConnection;
    sqlcdsJoins: TSQLClientDataSet;
    sqlcdsJoinsempno: TBCDField;
    sqlcdsJoinsename: TStringField;
    sqlcdsJoinsjob: TStringField;
    sqlcdsJoinsmgr: TBCDField;
    sqlcdsJoinshiredate: TSQLTimeStampField;
    sqlcdsJoinsal: TBCDField;
    sqlcdsJoinscomm: TBCDField;
    sqlcdsJoinsdname: TStringField;
    sqlcdsJoinsloc: TStringField;
    sqldsMGRS: TSQLDataSet;
    sqldsLocs: TSQLDataSet;
    sqldsMGRByENO: TSQLDataSet;
    sqldsMGRByENAME: TSQLDataSet;
    SQLMonitor1: TSQLMonitor;
    sqldsExecSQL: TSQLDataSet;
    procedure DataModuleCreate (Sender: TObject);
    procedure DataModuleDestroy (Sender: TObject);
    procedure sqlcdsJoinsMGRGetText (Sender: TField; var Text: String;
      DisplayText: Boolean);
    procedure sqlcdsJoinsMGRSetText (Sender: TField; const Text: String);
    procedure sqlcdsJoinsAfterPost (DataSet: TDataSet);
    procedure sqlcdsJoinsBeforeUpdateRecord (Sender: TObject;
      SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
      UpdateKind: TUpdateKind; var Applied: Boolean);
    procedure SQLMonitor1LogTrace (Sender: TObject; CBInfo: pSQLTRACE);
  private
    { Private declarations }
    procedure UpdateData (SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
```

```

procedure InsertData (SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
procedure DeleteData (SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
public
    { Public declarations }
end;

var
    dmJoinDemos: TdmJoinDemos;

implementation

uses fClientMain, Variants;

{$R *.dfm}

procedure TdmJoinDemos.DataModuleCreate (Sender: TObjedt;
begin
    sqlcJoins.Connected := True;
    sqlcdsJoins.Active := True;
    sqldsMGRByENAME.Prepared := True;
    sqldsMGRByENO.Prepared := True;
end;

procedure TdmJoinDemos.DataModuleDestroy (Sender: TObjedt;
begin
    sqldsMGRByENO.Prepared := False;
    sqldsMGRByENAME.Prepared := False;
    sqlcJoins.Connected := False;
end;

procedure TdmJoinDemos.sqlcdsJoinsMGRGetText (Sender: TField;
    var Text: String; DisplayText: Boolean
begin
    if (Sender.AsString <> )' then
        begin
            sqldsMGRByENO.Params.ParamByName ('EMPNO').Value := Sender.Value;
            try
                sqldsMGRByENO.Active := True;
                Text := sqldsMGRByENO.Fields[0].Value;
            finally
                sqldsMGRByENO.Active := False;
            end;
        end;
    end;

```

```

end;

procedure TdmJoinDemos.sqlcdsJoinsMGRSetText (Sender: TField;
  const Text: String);
begin
  sqlcdsMGRByENAME.Params.ParamByName ('ENAME').Value := Text;
  try
    sqlcdsMGRByENAME.Active := True;
    Sender.Value := sqlcdsMGRByENAME.Fields[0].Value;
  finally
    sqlcdsMGRByENAME.Active := False;
  end;
end;

procedure TdmJoinDemos.sqlcdsJoinsAfterPost (DataSet: TDataSet;
begin
  frmJoinMain.LabeledEdit1.Text := IntToStr(sqlcdsJoins.ChangeCount);
end;

procedure TdmJoinDemos.sqlcdsJoinsBeforeUpdateRecord (Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean;
begin
  frmJoinMain.cdsBKD.Data := TClientDataSet(DeltaDS).Data;
  case UpdateKind of
    ukModify :
      UpdateData (SourceDS, DeltaDS;
    ukInsert :
      InsertData (SourceDS, DeltaDS;
    ukDelete :
      DeleteData (SourceDS, DeltaDS;
  end;
  Applied := True;
end;

procedure TdmJoinDemos.SQLMonitor1LogTrace (Sender: TObject;
  CBInfo: pSQLTRACEDesc;
begin
  frmJoinMain.mmLogs.Lines.Add (CBInfo.pszTrace) ;
end;

procedure TdmJoinDemos.DeleteData (SourceDS: TDataSet;
  DeltaDS: TCustomClientDataSet

```



```

const
  sDeleteEMP = 'delete from EMP where EMPNO = :OLD_EMPNO';
  sDeleteDEPT = 'delete from DEPT where DEPTNO = :OLD_DEPTNO';

procedure DeleteEMP;
begin
  sqlDS.ExecSQL.CommandText := sDeleteEMP;
  sqlDS.ExecSQL.Params.ParamByName('OLD_EMPNO').Value :=
    DeltaDS.FieldByName('EMPNO').Value;;
  sqlDS.ExecSQL.ExecSQL(False);
end;

procedure DeleteDEPT;
begin
  sqlDS.ExecSQL.CommandText := sDeleteDEPT;
  sqlDS.ExecSQL.Params.ParamByName('OLD_DEPTNO').Value :=
    DeltaDS.FieldByName('DEPTNO').Value;;
  sqlDS.ExecSQL.ExecSQL(False);
end;
begin
  DeleteEMP;
  DeleteDEPT;
end;

procedure TdmJoinDemos.InsertData(SourceDS: TDataSet;
  DeltaDS: TCustomClientDataSet)
const
  sInsertEMP = 'insert into EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,
  DEPTNO) values
  (:EMPNO, :ENAME, :JOB, :MGR, :HIREDATE, :SAL, :COMM, :DEPTNO)';
  sInsertDEPT = 'insert into DEPT (DEPTNO, DNAME, LOC) values
  (:DEPTNO, :DNAME, :LOC)';

  procedure InsertEMP;
  begin
  end;

  procedure InsertDEPT;
  begin
  end;
begin
  InsertEMP;
  InsertDEPT;
end;

```

```
procedure TdmJoinDemos.UpdateData (SourceDS: TDataSet;
DeltaDS: TCustomClientDataSet);
const
    sUpdateEMP = 'update EMP set EMPNO = :EMPNO, ENAME = :ENAME, JOB = :JOB, MGR = :MGR, HIREDATE = :HIREDATE, SAL = :SAL, COMM = :COMM where EMPNO = :OLD_EMPNO ';
    sUpdateDEPT = 'update DEPT set DEPTNO = :DEPTNO, DNAME = :DNAME, LOC = :LOC where DEPTNO = :OLD_DEPTNO';

procedure UpdateEMP;
var
    iCount : Integer;
    sFieldName : String;
    nValue : Variant;
begin
    sqldsExecSQL.CommandText := sUpdateEMP;
    for iCount := 0 to DeltaDS.FieldCount - 1 do
        begin
            nValue := DeltaDS.Fields[iCount].NewValue;
            sFieldName := DeltaDS.Fields[iCount].FieldName;
            if (Assigned (sqldsExecSQL.Params.FindParam (sFieldName))) then
                begin
                    if (not VarIsEmpty (nValue)) then
                        sqldsExecSQL.Params.ParamByName (sFieldName).Value := nValue
                    else
                        sqldsExecSQL.Params.ParamByName (sFieldName).Value := DeltaDS.Fields[iCount].OldValue;
                end;
            end;
            sqldsExecSQL.Params.ParamByName ('OLD_EMPNO').Value := DeltaDS.FieldByName ('EMPNO').OldValue;
            sqldsExecSQL.ExecSQL (False);
        end;

procedure UpdateDEPT;
var
    iCount : Integer;
    sFieldName : String;
    nValue : Variant;
begin
    sqldsExecSQL.CommandText := sUpdateDEPT;
    for iCount := 0 to DeltaDS.FieldCount - 1 do
        begin
            nValue := DeltaDS.Fields[iCount].NewValue;
```

```

sFieldName := DeltaDS.Fields[iCount].FieldName;
if (Assigned (sqlcsExecSQL.Params.FindParam (sFieldName))) then
begin
  if (not VarIsEmpty (nValue)) then
    sqlcsExecSQL.Params.ParamByName (sFieldName).Value := nValue
  else
    sqlcsExecSQL.Params.ParamByName (sFieldName).Value := DeltaDS.Fields
[iCount].OldValue;
  end;
end;
sqlcsExecSQL.Params.ParamByName ('OLD_DEPTNO').Value := DeltaDS.FieldByName
('DEPTNO').OldValue;
sqlcsExecSQL.ExecSQL (False);
end;
begin
  UpdateEMP;
  UpdateDEPT;
end;
end.

```

首先让我们讨论 `sqlcsJoins` 组件如何实现它的 `BeforeUpdateRecord` 事件处理函数。在 `sqlcsJoinsBeforeUpdateRecord` 函数中，参数 `DeltaDS` 包含的是传递来的经过用户修改而需要更新回数据源的数据。因此 `sqlcsJoinsBeforeUpdateRecord` 先把 `DeltaDS` 指定给主窗体中的 `TClientDataSet` 以显示 `sqlcsJoins` 的 `Delta` 数据。接着，`sqlcsJoinsBeforeUpdateRecord` 根据目前这个需要修改的记录的最新种类 `UpdateKind` 来判断它是需要修改的数据，还是需要删除的数据或是用户新增的数据，再调用不同的函数来处理这个记录。

由于更新 `Join` 数据的操作比较复杂，因此我们需要讨论 `UpdateData` 方法是如何实现的。`UpdateData` 方法需要从 `sqlcsJoins` 的 `Delta` 特性值中一一取出分属于每一个数据表的字段值，然后判断这个字段是否被用户修改过。判断的方法是检查 `TField` 对象的 `NewValue` 特性值是否有值。如果字段被修改过，那么 `NewValue` 便有新的值；如果字段没有被修改过，那么 `NewValue` 便是空白的。对于没有被修改过的字段，就访问 `TField` 对象的 `OldValue` 以取得字段的原始数值。最后把每一个字段值代入 `Update SQL` 语句的动态参数中，再调用 `TSQLDataSet` 的 `ExecSQL` 方法执行 `SQL` 语句并且更新数据。由于当程序代码进入 `TSimpleDataSet/TClientDataSet` 的 `ApplyUpdates` 方法后，`DataSnap` 会自动进入数据库事务管理，因此我们可以很放心地在这里同时更新数个数据表，最后使用 `TSQLDataSet` 直接执行 `SQL` 语句也是很有效率的方式。

因此在上面的 `UpdateData` 方法中先声明了两个 `Update SQL` 语句，`sUpdateEMP` 和 `sUpdateDEPT`，接着分别调用 `UpdateEMP` 和 `UpdateDEPT` 内嵌方法来更新 `EMP` 和

DEPT数据表。

UpdateEMP方法首先将sUpdateEMP SQL语句指定给数据模块中的 TSQLDataSet 组件sqldsExecSQL，然后进入循环取出字段名并且通过 NewValue特性值判断此字段值是否经过修改，接着使用字段名在动态 SQL语句中找到相同名称的动态参数，最后把判断过的字段值代入动态参数中并且执行 sqldsExecSQL的ExecSQL方法更新数据。UpdateDEPT方法也使用一样的技巧。但是，读者需要注意的是 UpdateEMP和UpdateDEPT方法对于键值字段都必须代入原始的字段值，这意味着这个范例程序不允许用户修改这两个数据表的键值字段的值。

现在编译并且执行这个范例程序，更新数个记录之后再点击主窗体中的“ApplyUpdates”按钮把Join的数据更新回 Oracle数据库，那么读者应该可以看到类似于图6 18和图6 19的两个画面，在客户端修改的数据的确可以通过 TSimpleDataSet 的BeforeUpdateRecord事件处理函数分别更新回两个不同的数据表中。

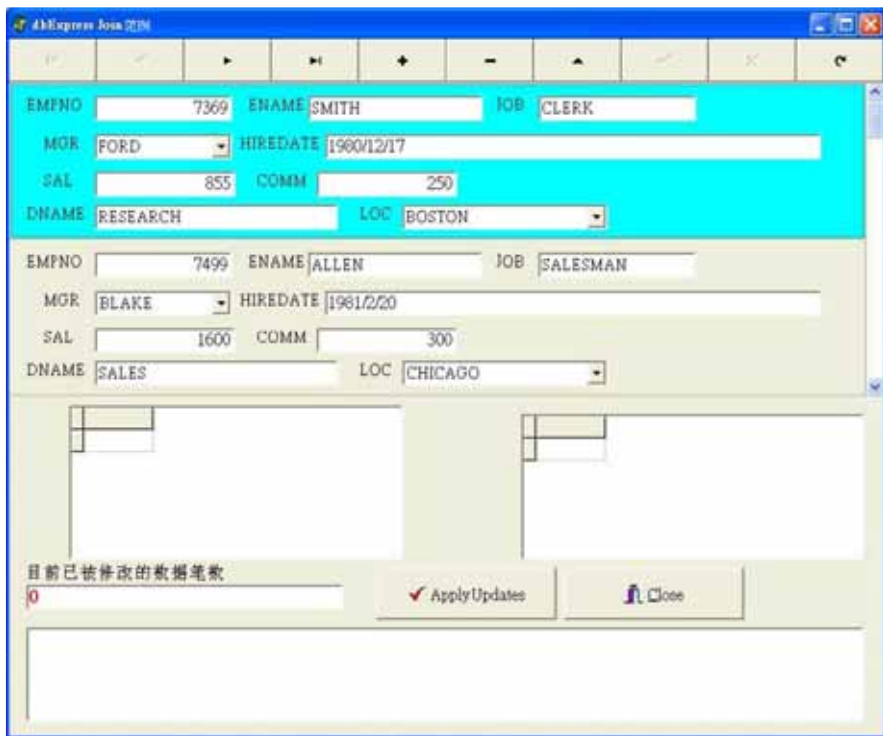


图6-18 执行范例程序并且准备将数据更新回数据源中

这个范例程序实现了将数据更新回数据源的功能，但是尚未实现在数据源中删除和新增数据的功能。事实上执行删除和新增数据的方式与前面讨论的更新数据使用的技巧是一样的，因此这两个比较简单的功能就留给读者当作练习。不过在新增数据方面读者要注意的是，由于许多数据表在定义时可能会规定一些字段必须有值，

因此在处理Join的数据并且新增回数据源时，对于这些字段一定要提供字段值，否则可能发生错误。

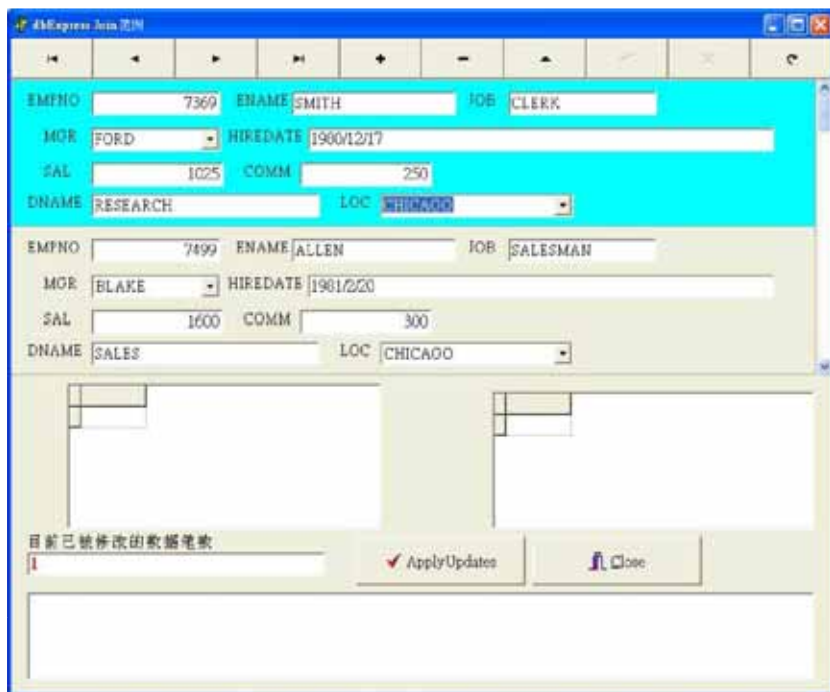


图6-19 执行范例程序并且成功地将数据更新回数据源中

6 4 结论

本章讨论了TDataSetProvider组件并且说明这个组件在dbExpress应用程序中可以提供的功能。虽然 Delphi 6提供了TSQLClientDataSet组件，Delphi 7提供了TSimpleDataSet组件以方便程序员开发dbExpress应用程序。但是程序员如果想要完整地控制dbExpress中访问和修改数据的行为并且得到最好的性能，那么使用TClientDataSet加上TDataSetProvider组件将是最好的组合。

虽然使用TClientDataSet和TDataSetProvider比使用TSQLClientDataSet或是TSimpleDataSet麻烦了一点，但是TClientDataSet加TDataSetProvider提供的弹性却比使用TSQLClientDataSet/TSimpleDataSet好得多了。在本章介绍了TDataSetProvider的重要概念、方法和事件处理函数之后，在稍后的小节中使用了TDataSetProvider的知识来解决将Join数据更新回多个数据表的问题。在充分了解了TDataSetProvider以及如何通过这些知识来处理复杂的数据之后，读者便可以使用这两个组件来处理任何复杂的数据需求了。

第7章 dbExpress和Web应用程序

对于开发Web应用系统来说，dbExpress是非常适合使用的数据库访问技术。除了因为dbExpress可以同时运行在Windows和Linux上之外，由于大部分的Web应用程序都是查询数据或是处理静态数据和图形，因此dbExpress的单向、只读特性正是这种应用最有效率的数据访问技术。

本章讨论的内容除了说明Delphi/Kylix的WebBroker技术让读者了解如何使用Delphi/Kylix提供的Web功能来开发Web应用程序之外，主要的重点是说明如何结合WebBroker和dbExpress技术来开发以数据为中心的Web应用系统。本章在说明了WebBroker的概念和技术之后，便开始结合dbExpress一步一步地告诉读者如何在WebBroker中使用dbExpress来快速地访问数据并且根据数据产生返回的HTML结果。在读者了解了WebBroker和在其中使用dbExpress要注意的事项之后，最后会说明如何在Delphi的项目组中建立ISAPI类型的Web应用程序，以得到更有性能的Web应用程序。

7.1 Delphi的WebBroker技术

Delphi 7提供了数种不同的技术让程序员开发Web应用系统，例如从Delphi 3便出现的WebBroker，Delphi 5开发的InternetExpress，以及Delphi 6/7的WebSnap技术等。基本上WebBroker、InternetExpress和WebSnap都有不同的应用，而WebBroker也是其他两种Web技术的基础，只是InternetExpress和WebSnap都各自增加了许多功能帮助程序员更方便地开发Web应用系统。而InternetExpress除了允许程序员开发处理静态数据的Web应用系统之外，也可以让程序员开发能够修改数据的Web应用系统。WebSnap则是Borland开发的跨平台Web技术，WebSnap不但能够在Windows平台上使用，Borland在Kylix 3中也已经提供了在Linux平台上执行的WebSnap。此外，WebSnap还使程序员能够使用脚本语言（例如VBScript和JavaScript）来开发Web应用程序，让程序员能够拥有更大的弹性。

由于本书不是专门讨论如何使用Delphi/Kylix开发Web应用系统的书籍，因此无法详细说明InternetExpress和WebSnap的技术。本章的内容将以WebBroker配合dbExpress来说明如何在WebBroker中使用dbExpress来快速提供数据处理能力。由于WebBroker已经可以同时运行在Windows和Linux平台中，而dbExpress是Borland用来支持同时在Windows和Linux平台中访问数据的引擎，因此程序员可以结合Web

Broker以及dbExpress技术在Windows和Linux平台中开发出最具效率的Web应用系统。

7.2 使用dbExpress开发Web应用程序

从本小节开始将使用一个范例来一步一步地使用 dbExpress开发范例Web应用程序。在这个范例中将会使用 dbExpress来查询和修改数据,让读者了解如何在Web应用程序中使用dbExpress,以及一些必须注意的技术细节。

首先本范例使用了一个HTML模板文件,图7-1是此模板文件的画面。此HTML模板使用了数个HTML控件以及WebBroker的定制标记,例如<#QueryResult>和<#SystemTime>,在随后的小节中将展示如何使用dbExpress查询数据来取代这些定制标记。

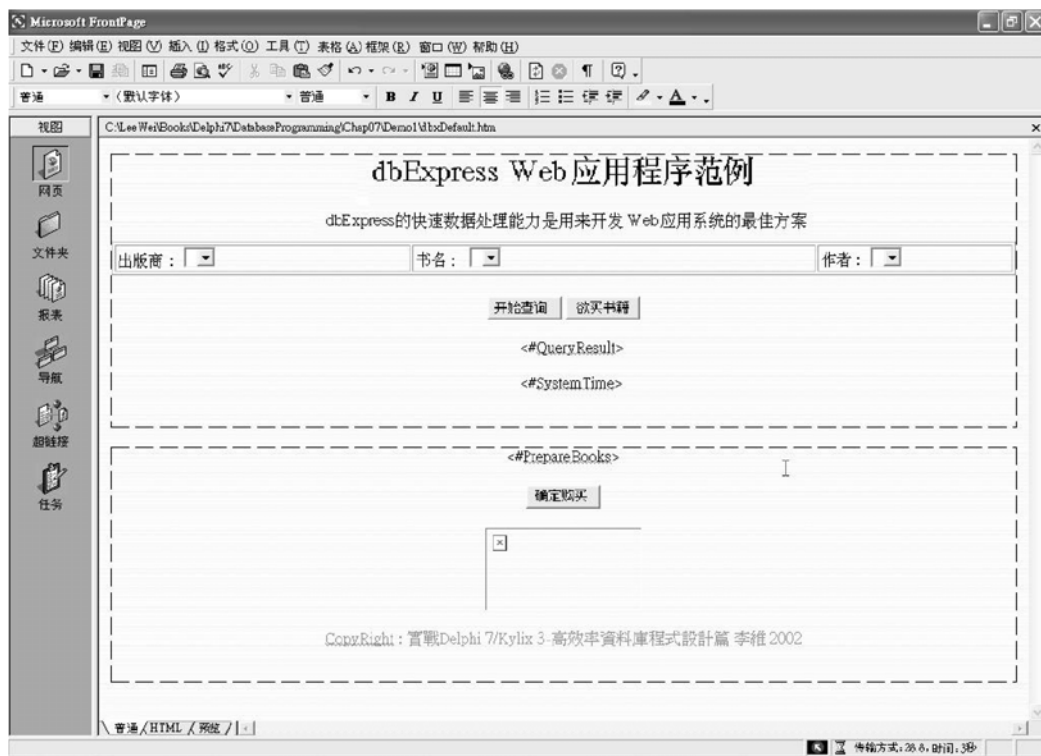


图7-1 范例Web应用程序使用的HTML模板

读者可以在本书的光盘中找到这个HTML模板文件,dbxDefault.HTM。

步骤 1: 建立Web应用程序

首先让我们开发一个CGI类型的Web应用程序,因为CGI应用程序比较容易调试和使用。在完成了这个CGI范例应用程序的开发之后,在稍后的小节中会说明如何

把这个CGI应用程序转换为 ISAPI类型的Web应用程序，以取得较好的 Web应用程序性能。

首先点击File New Other菜单，再选择建立 CGI类型的Web应用程序，如图 7 2所示。

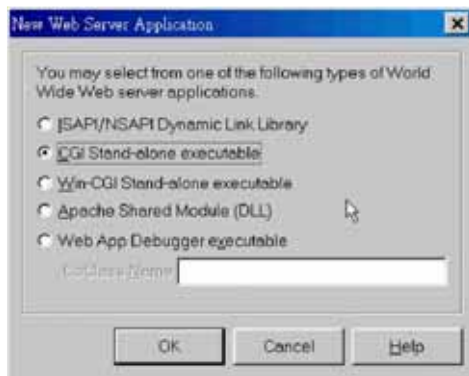


图7-2 建立CGI类型的Web应用程序

接着在空白的WebModule中放入一个TPageProducer、一个TSQLConnection和一个TSQLDataSet组件。将TSQLConnection设置为连接到范例数据库 D7Books，再将TSQLDataSet设置为连接到此TSQLConnection并且设置它的CommandText特性值为select * from BOOKS，如图 7 3所示。

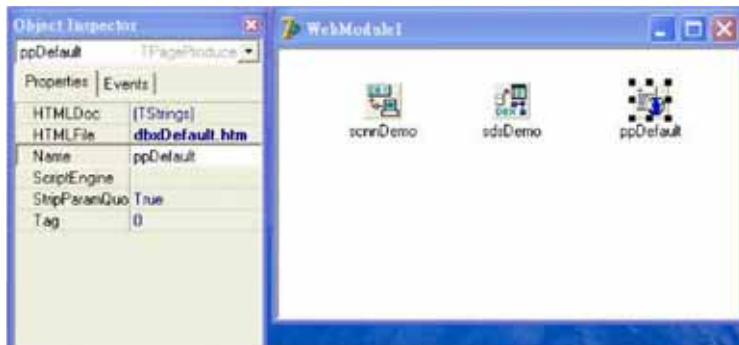


图7-3 在WebModule中放入TSQLConnection、TSQLDataSet和TPageProducer组件

首先我们希望这个范例 Web应用程序能够返回图 7 1所示的HTML模板文件，因此让我们双击WebModule以激活WebModule的WebAction编辑器。然后点击上方的Add New按钮在其中加入一个WebActionItem，设置此WebActionItem的Name特性值为waDfault，并且设置它的Producer特性值为WebModule中的ppDefault，以返回ppDefault的内容，最后设置它的Default特性值为True，代表这是此Web应用程序的默认主页内容，如图 7 4所示。

由于默认返回的主页内容是 ppDefault的内容，因此我们必须设置 ppDefault的

HTMLFile特性值为dbxDefault.HTM，以返回HTML模板文件。



图7-4 为WebModule定义默认WebAction

现在请编译此范例 Web 应用程序，并且把产生的 EXE 应用程序拷贝到你的 Web 虚拟目录之下，然后激活浏览器，在网址栏中输入执行此 Web 应用程序的 URL，那么 Web Service 便会执行我们的范例 Web 应用程序，并且返回 HTML 模板文件内容。例如图 7 5 便是在 IE 中执行范例 Web 应用程序之后看到的画面，现在范例 Web 应用程序已经可以正确地返回本章使用的 HTML 模板文件的内容。



图7-5 在浏览器中看到范例 Web 应用程序返回的默认网页

步骤 2: 处理定制标记

在前面使用的模板 HTML 文件中使用了数个定制标记, 包含了 `<#SystemTime>` 以及 `<#QueryResult>`。这两个定制标记在 Web 应用程序执行时将被目前的服务器系统时间以及用户查询的书籍信息取代。现在就让我们说明如何处理这些定制标记。

当 WebBroker 返回 HTML 结果时, 如果发现其中有定制标记, 就会触发 TPageProducer 的 OnHTMLTag 事件处理函数。因此程序员可以在 OnHTMLTag 事件处理函数中用特定的字符串来取代任何定制标记。由于在步骤 1 中是用 ppDefault 加载模板 HTML 文件并且返回此文件的内容, 因此我们只需要为这个 TPageProducer 定义 OnHTMLTag 事件处理函数, 在其中取代 `<#SystemTime>` 和 `<#QueryResult>`。由于 `<#QueryResult>` 是稍后用户查询的书籍信息, 因此我们先用服务器时间取代 `<#SystemTime>`。下面就是处理 `<#SystemTime>` 定制标记的程序代码:

```
procedure TwmdbxDemo.wmDemolwaDefaultAction (Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    Response.Content := ppDefault.Content;
    Handled := True;
end;

procedure TwmdbxDemo.ppDefaultHTMLTag (Sender: TObject; Tag: TTag;
const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    case Tag of
        tgCustom :
            begin
                if (TagString = 'SystemTime') then
                begin
                    ReplaceText :=
                        '<h4 align="center"><font color="#008080"> 现在时间: ' + DateTimeToStr
                        (Now) + '</font></h4>';
                    Exit;
                end;
            end;
    end;
end;
```

上面的程序代码在 ppDefault 的 OnHTMLTag 事件处理函数中判断现在处理的是否是定制标记, 如果是而且这个标记是 `<#SystemTime>` 的话, 就用目前的时间取代它。

完成了上面的程序代码之后, 现在如果再次编译和执行范例 Web 应用程序, 那么就可以在浏览器中看到如图 7-6 所示的画面。现在范例应用程序不但显示了模板 HTML 文件的内容, 也用目前的服务器时间取代了 `<#SystemTime>` 定制标记。

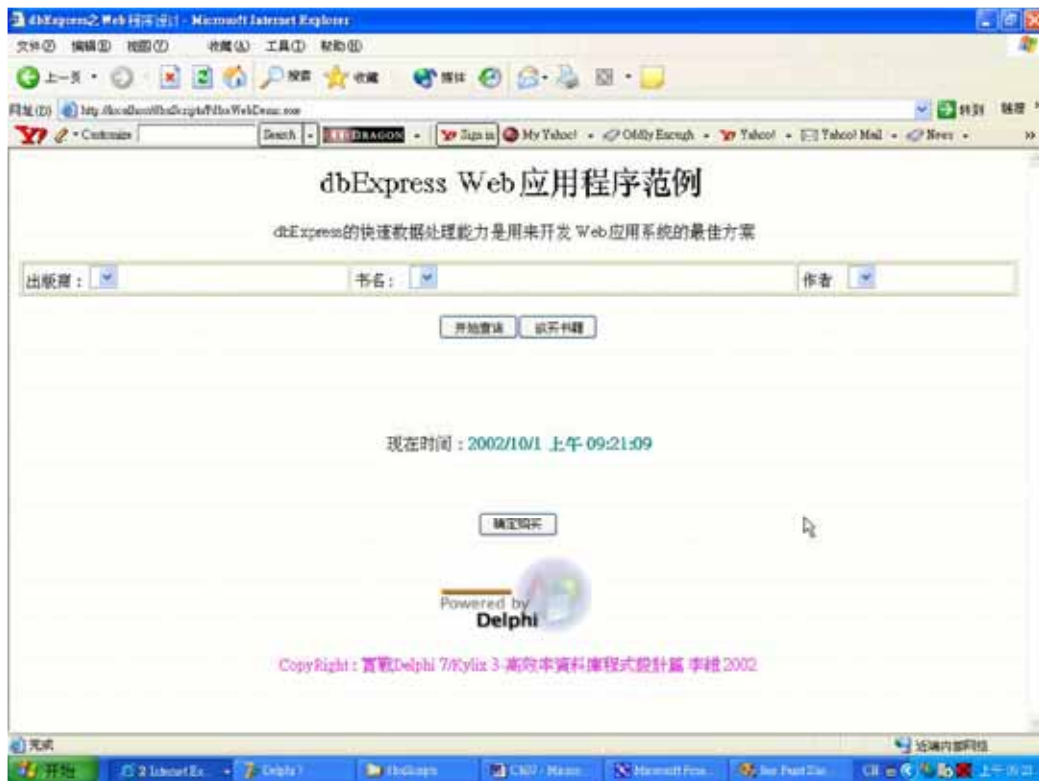


图7-6 用现在的服务器时间取代<#SystemTime>定制标记

步骤3：使用dbExpress处理定制标记

接下来的步骤就是使用 dbExpress 访问数据库中的数据，并且搭配使用定制标记来呈现模板 HTML 文件。现在希望能够在图 7 6 中的出版商下拉框中显示数据库中 PUBLISHERS 数据表中书籍出版商的名称，以便让用户能够选择要查询的出版商。当用户选择了一个出版商之后，再在书名下拉框中显示这个出版商出版的所有书籍的名称，此时又需要使用 dbExpress 到 BOOKS 数据表中查询数据了。

要在模板 HTML 中加入数据库中的信息，我们首先需要在模板 HTML 文件中加入一些定制标记，以便像前一小节处理 <#SystemTime> 定制标记一样进行处理。请使用 FrontPage 或是任何编辑器开启模板 HTML 文件，然后在其中加入如下的定制标记：

```
<table border="1" width="100%">
  <tr>
    <td width="22%"> 出版商      :&nbsp; <select size="1"
name="cbPublishers">
    &nbsp;<#Publishers> ◀—————
    &nbsp;</select>&nbsp;&nbsp;&nbsp;&nbsp;</td>
    <td width="25%"> 书名 : &nbsp;&nbsp;&nbsp;<select size="1" name="cbBook">
```

定制卷标

```

        &nbsp;&nbsp;&nbsp;<#Books>
        &nbsp;&nbsp;&nbsp;</select></td> <td width="53%">
        作者 :&nbsp;&nbsp;&nbsp;<select size="1" name="cbAuthor">
        &nbsp;&nbsp;&nbsp;<#Authors>
        &nbsp;&nbsp;&nbsp;</select></td>
    </tr>
</table>

```

定制卷标

定制卷标

上面的程序代码在模板 HTML文件中加入了 `<#Publishers>`、`<#Books>`和`<#Authors>`这三个定制标记。现在我们需要访问数据表中的数据并且用这些数据取代这些定制标记。

由于需要从数据库中取出数据，因此请在 WebModule中加入 TSQLConnection 组件，将它连接到范例数据库 D7Books，加入 TSQLDataSet 组件，设置它的 Name 特性值为 sdsPublishers，设置它的 CommandText 为 `select NAME, VID from PUBLISHERS where PKIND = 'BK'`，从 PUBLISHERS 数据表中选择出书籍的出版商。再放入另外一个 TSQLDataSet 组件，设置它的 Name 特性值为 sdsDemo，设置它的 CommandText 为 `select * from BOOKS`，此时 WebModule 如图 7-7 所示。



图7-7 在WebModule中继续加入TSQLDataSet以查询出版商信息

加入了这些 dbExpress 组件之后，现在我们仍然在 ppDefault 的 OnHTMLTag 事件处理函数中处理这些新加入的定制标记，就像处理 `<#SystemTime>` 一样。下面就是修改后的 ppDefault 的 OnHTMLTag 事件处理函数。请注意在新的程序代码中加入了处理 `<#Publishers>`、`<#Books>` 和 `<#Authors>` 这三个定制标记的程序代码。

```

procedure TwmdbxDemo.ppDefaultHTMLTag (Sender: TObject; Tag: TTag;
const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    case Tag of
        tgCustom :
            begin
                if (TagString = 'SystemTime') then
                    begin

```



```

...
end;

if (TagString = PUBLISHERTAG) then
begin
  ReplaceText := DoGetPublishers
  Exit;
end;

if (TagString = BOOKTAG) then
begin
  ReplaceText := DoGetBooks;
  Exit;
end;

if (TagString = AUTHORTAG) then
begin
  ReplaceText := DoGetAuthors;
  Exit;
end;
end;
end;
end;

```

在上面的程序代码中，当 `ppDefault` 的 `OnHTMLTag` 事件处理函数处理到 `<#Publishers>` 定制标记时，就调用 `DoGetPublishers` 函数来处理。同样，当处理到 `<#Books>` 定制标记时就调用 `DoGetBooks` 函数来处理，而对于 `<#Authors>` 标记调用 `DoGetAuthors` 函数。这些函数都使用 `dbExpress` 到范例数据库中取得数据，再用取得的数据取代这些定制标记，而如何在 Web 应用程序中使用 `dbExpress` 来访问数据就是本章的重点。

首先让我们看看 `DoGetPublishers` 如何从 `PUBLISHERS` 数据表中取出数据。下面就是 `DoGetPublishers` 函数的程序代码：

```

function TwmdbxDemo.DoGetPublishers : String;
var
  bFirst : Boolean;
  aTD: TTransactionDesc;
begin
  aTD.TransactionID := 1;
  aTD.IsolationLevel := xilREADCOMMITTED;

  scnnDemo.StartTransaction (aTD);
  try

```

```
with sdsPublishers do
begin
    Prepared := True;
    Open;
    bFirst := True;
    while not Eof do
    begin
        if (bFirst) then
        begin
            Result := Result + '<option selected>' + Fields[0].AsString + '</option>';
            bFirst := False;
        end
        else
        begin
            Result := Result + '<option>' + Fields[0].AsString + '</option>';
        end;
        Next;
    end;
    Prepared := False;
    Close;
end;
except
on E: Exception do
begin
    Result := E.Message;
    scnnDemo.Rollback (aTD);
end;
end;

if scnnDemo.InTransaction then
    scnnDemo.Commit (aTD);
end;
```

上面的程序代码首先声明了一个 dbExpress 的事务变量，接着设置它的事务 ID，并且设置事务使用的隔离级别为 READCOMMITTED。要在 Web 应用程序中使用 dbExpress 访问数据，首先必须让数据库进入事务模式。

注意，MySQL 4.0 之前不支持事务，因此我们可以在上面的程序代码中多加一行 `if (TransactionsSupported) then`，以判断读者使用的数据库是否支持事务，再调用 `StartTransaction`。

接着先准备sdsPublishers组件，再开启sdsPublishers组件从PUBLISHERS数据表中取得所有书籍出版商的数据。然后进入循环中一一取得每一个出版商的数据，把出版商的名称组成HTML的合法字符串形式，再指定给Result变量做为DoGetPublishers函数的返回值。最后，如果在访问数据时发生错误，那么调用TSQLConnection的Rollback，再返回发生的错误原因，否则就调用Commit确保数据访问完毕。在开发Web应用程序时，虽然只是读取数据，但是我们最好还是激活一个事务来封装这个读取流程，这样不但可以确保数据的完整性，也可以减少在Web应用程序中发生问题的可能性。

上面的程序代码事实上是开发者在使用dbExpress开发Web应用程序时必须使用的程序代码模板，这就是说为了避免在Web应用程序中发生问题，程序员必须使用与上面的程序代码类似的方式来访问数据。

为什么在Web应用程序中访问数据容易发生问题呢？这是因为Web应用程序可能同时被多个线程访问，例如稍后要开发的ISAPI应用程序便是一种多线程的Web应用程序。不过不是所有数据库客户端访问软件都是多线程安全的(Thread Safe)，因此让每一个访问线程使用独立的数据库连接并且使用事务进行隔离是比较安全的，也可以克服非多线程安全的数据库客户端软件的风险。

完成了DoGetPublishers函数之后，如果我们实现空白的DoGetBooks和



图7-8 现在范例Web应用程序在浏览器中可以显示所有出版商信息

DoGetAuthors函数，然后编译并且执行范例 Web应用程序，那么就可以在浏览器中看到类似于图 7 8的画面，现在在模板 HTML的出版商下拉框中就可以看到目前在 PUBLISHERS数据表中所有出版书籍的出版商的名称了。

步骤 4：使用dbExpress处理Web查询数据

在步骤3中我们已经学习了如何在 Web应用程序中使用 dbExpress访问数据，剩下的步骤就是混合运用 Web程序技术以及 dbExpress来完成这个范例 Web应用程序了。现在用户可以在浏览器中选择出版商的名称来查询书籍数据了，要让范例 Web应用程序能够根据出版商名称查询书籍数据，我们必须为范例 Web应用程序加入一个新的WebAction来处理查询书籍数据的 HTTP Post行动。

由于前面的步骤都是以默认的 WebAction作为处理HTTP请求的处理对象，现在我们需要加入一个新的 WebAction来处理查询书籍的动作，因此请先修改模板 HTML文件，在Form的HTML标记处为 action标记加入新的 QueryPath， /dbxQuery，如下所示：

```
<form method="POST" action="/dbxScripts/PdbxWebDemo1.exe/dbxQuery ">
  <h1 align="center"><font color="#0000FF">dbExpress
  ...
<p align="center"> </p>
</form>
```

接着在 WebModule中加入一个新的 TSQLDataSet，设置它的 Name特性值为 sdsPublishersByName，CommandText为 select VID from PUBLISHERS where NAME = :NAME，以出版商名称查询出版商 ID，如图 7 9所示。

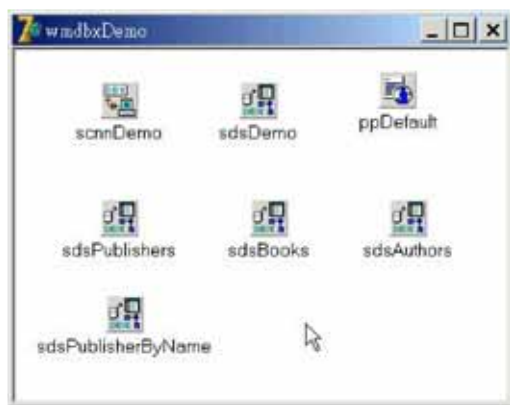


图7-9 在WebModule中加入另一个TSQLDataSet以查询作者、书籍等信息

接着实现 DoGetBooks，程序代码如下：

```
function TwmdbxDemo.DoGetBooks : String;
var
  bFirst : Boolean;
```

```

sVID : String;
sBookName : String;
aTD: TTransactionDesc;
begin
Result := '<option>' 尚无数据' + '</option>';//';
if (sPublisher = '') then
    exit
else
    Result := '';

aTD.TransactionID := 1;
aTD.IsolationLevel := xilREADCOMMITTED;

scnnDemo.StartTransaction(aTD);
try
    sVID := GetPublisherVID;
    sdsBooks.ParamByName('VID').Value := sVID;
    with sdsBooksdo
    begin
        Prepared := True;
        Open;
        bFirst := True;
        while not Eof do
        begin
            sBookName := FieldByName('BOOKNAME').AsString;
            if (bFirst) then
            begin
                Result := Result + '<option selected>' + sBookName + '</option>';
                bFirst := False;
            end
            else
            begin
                if (sBook = sBookName) then
                    Result := Result + '<option selected>' + sBookName + '</option>'
                else
                    Result := Result + '<option>' + sBookName + '</option>';
                end;
            Next;
        end;
        Prepared := False;
        Close;
    end;
except

```

```

on E: Exception do
begin
    Result := E.Message;
    scnnDemo.Rollback (aTD);
end;

end;

if scnnDemo.InTransaction then
    scnnDemo.Commit (aTD);
end;

```

上面的程序代码首先调用 **GetPublisherVID**方法根据出版商的名称取得出版商的ID，接着以出版商的ID为键值，使用 **sdsBooks**从 **Books**数据表中选择出此出版商出版的所有书籍，然后如同前面的 **DoGetPublishers**一样把所有书籍的名称以 **HTML**的格式返回。

而 **GetPublisherVID**则是以出版商的名称做为查询的键值，并且返回出版商的ID。出版商的名称是从浏览器的 **HTTP Post**中取出的，因此我们先修改 **ppDefault**的 **OnHTMLTag**事件处理函数，调用 **ParseContentFields**从 **HTTP**请求中解析出必要的信息，例如用户选择的出版商名称等：

```

procedure TwmdbxDemo.wmDemo1waDefaultAction (Sender: TObject;
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    ParseContentFields (Request);
    Response.Content := ppDefault.Content;
    Handled := True;
end;

```

ParseContentFields函数通过 **TWebRequest**对象的 **ContentFields**特性值取得用户选择的出版商名称、书籍名称和作者名称，程序代码如下：

```

procedure TwmdbxDemo.ParseContentFields (Request: TWebRequest);
begin
    sPublisher := Request.ContentFields.Values['cbPublisher'];
    sBook := Request.ContentFields.Values['cbBook'];
    sAuthor := Request.ContentFields.Values['cbAuthor'];
end;

```

上面程序代码中的 ‘**cbPublisher**’、‘**cbBook**’ 和 ‘**cbAuthor**’ 是模板 **HTML**文件中三个下拉框的名称，我们可以通过 **TWebRequest**的 **ContentFields**访问用户在这三个下拉框中选择的信息。

DoGetBooks调用的 **GetPublisherVID**使用从 **TWebRequest**中取出的 **sPublisher**出版商名称通过数据模块中的 **sdsPublisherByName**组件查询出版商ID：

```

function TwmdbxDemo.GetPublisherVID: String;

```

```
begin
    sdsPublisherByName.Active := False;
try
    sdsPublisherByName.ParamByName('NAME').Value := sPublisher;
    sdsPublisherByName.Active := True;
    Result := sdsPublisherByName.FieldByName('NAME').Value;
finally
    sdsPublisherByName.Active := False;
end;
end;
```

现在编译并且执行范例 Web 应用程序，图 7 10 是此时范例 Web 应用程序在浏览器中的画面。从图 7 10 中可以看到，当用户选择了出版商名称（旗标）时便可以查询到由这个出版商出版的所有书籍，并且显示在书名下拉框中。

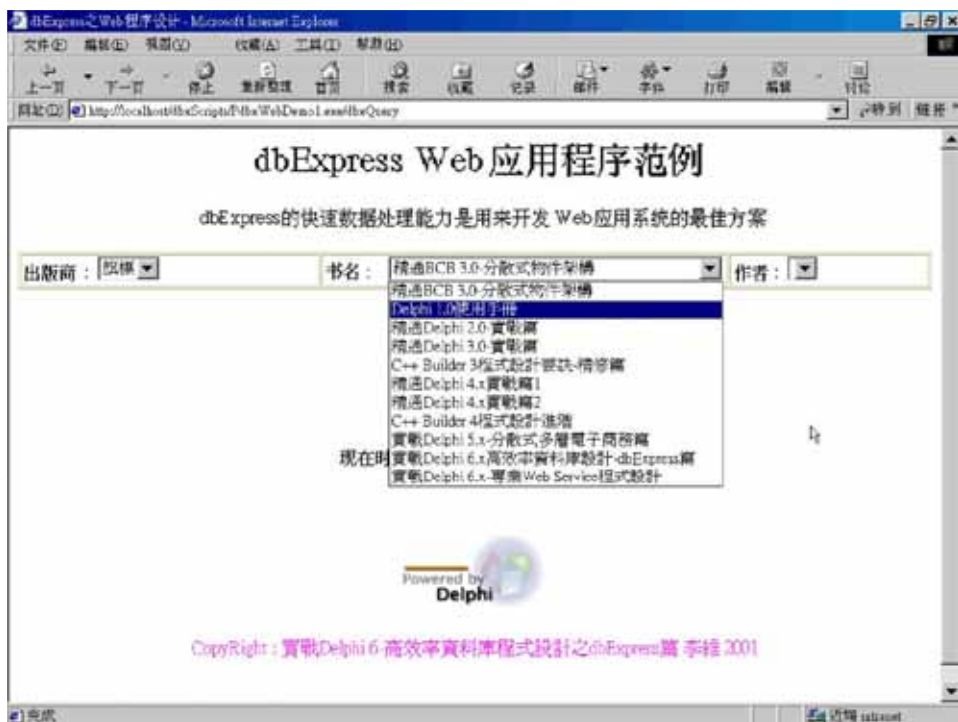


图7-10 范例Web应用程序在浏览器中根据出版商查询书籍信息

作者下拉框的技巧和 DoGetBooks 函数一样，就留给各位读者来完成了。现在再让我们看看如何从图 7 10 继续开发，让用户能够在书名下拉框中选择一本书来查询此书的详细数据，并且显示在浏览器中。

我们准备把书籍的详细数据显示在定制标记 <#QueryResult> 中，因此我们只需要在 ppDefault 处理到 <#QueryResult> 时用书籍的详细数据取代它即可。因此我们可以

先在ppDefault的OnHTMLTag事件处理函数中加入处理 <#QueryResult>定制标记的程序代码，代码如下：

```
if (TagString = QUERYRESULTTAG) then
begin
  ReplaceText := DoGetQueryResult;
  Exit;
end
```

接着就需要实现 DoGetQueryResult函数了。DoGetQueryResult的实现也很简单，它会根据用户在书名下拉框中选择的书籍名称到 Books数据表中查询，再把此书的数据组成HTML的格式，然后返回。下面就是 DoGetQueryResult的实现程序代码：

```
function TwmdbxDemo.DoGetQueryResult: String;
const
  bgColor1 = '#ccccff';
  bgColor2 = '#ffffcc';
var
  sHeader : String;
  bgColor : string;
  sINUT002 : String;
  Row      : Integer;
  aTD: TTransactionDesc;
begin
  Result := '';
  aTD.TransactionID := 1;
  aTD.IsolationLevel := xilREADCOMMITTED;
  scnnDemo.StartTransaction (aTD);

  SetupQuerySQL;
  try
    sHeader := GetHeader;
    with sdsDemo do
    begin
      Open;
      Result := sHeader + SetupTableColumn;
      Row := 0;
      while not Eof do
      begin
        if (Row mod 2 = 0) then
          bgColor := bgColor1
        else
          bgColor := bgColor2;
        inc (Row);
```

```

Result := Result +
    '    <tr bgcolor="' + bgColor + '">'#13#10;

Result := Result +
    '        <td>' + Fields[0].AsString + '</td>'#13#10;
Result := Result +
    '        <td>' + Fields[1].AsString + '</td>'#13#10;
Result := Result +
    '        <td>' + Fields[2].AsString + '</td>'#13#10;
Result := Result +
    '        <td>' + Fields[3].AsString + '</td>'#13#10;
Result := Result +
    '        <td>' + Fields[4].AsString + '</td>'#13#10;
Result := Result +
    '        <td>' + Fields[5].AsString + '</td>'#13#10;

Result := Result +
    '    </tr>'#13#10;
Next;
end;

Prepared := False;
Close;
end;
Result := Result + '</table>'#13#10;
except
    on E: Exception do
    begin
        Result := E.Message;
        scnnDemo.Rollback (aTD) ;
    end;
end;

if scnnDemo.InTransaction then
    scnnDemo.Commit (aTD) ;
end;
end;

```

DoGetQueryResult首先激活dbExpress的事务，然后调用SetupQuerySQL根据用户选择的书名组成SQL语句，然后使用这个SQL语句到Books数据表中查询数据。最后根据查询到的数据组成HTML字符串并且返回，再Commit这个dbExpress事务。DoGetQueryResult使用的程序代码和前面的DoGetBooks函数非常类似。

DoGetQueryResult调用的SetupQuerySQL函数的实现如下所示：

```

procedure TwndbxDemo.SetupQuerySQL;

```

```

var
    sSQL : String;
    sVID : String;
    sAID : String;
begin
    sdsDemo.Active := False;
    sdsDemo.Prepared := False;

    sSQL := 'select * from Books ';
    if (sPublisher <> '') then
    begin
        sVID := GetPublisherVID;
        sSQL := sSQL + ' where VID = ' + ''' + sVID + ''';
    end;

    if (sBook <> '') then
        sSQL := sSQL + ' and BOOKNAME = ' + ''' + sBook + ''';

    if (sAuthor <> '') then
    begin
        sAID := GetAuthorAID;
        sSQL := sSQL + ' and AID = ' + ''' + sAID + ''';
    end;

    sdsDemo.CommandText := sSQL;
    sdsDemo.Prepared := True;
end;

```

SetupQuerySQL会根据TWebRequest中被解析出的出版商名称、书籍名称等信息组成正确的SQL语句以便从Books数据表中选择出用户查询的书籍。

除了SetupQuerySQL之外，DoGetQueryResult也调用了GetHeader函数。而GetHeader函数返回HTML的table标记，把书籍的信息封装在表格中。

```

function TwmdbxDemo.GetHeader: string;
begin
    Result :=
        '<table border="1" cellpadding="2" cellspacing="2">' + #13#10 +
        '  <tr bgcolor="silver">' + #13#10;
end;

function TwmdbxDemo.SetupTableColumn: String;
begin
    Result := Result +

```

```

        <td><b>' + 'ISBN' + '</b></td>'#13#10;
Result := Result +
        <td><b>' + 书籍编号' + '</b></td>'#13#10;
Result := Result +
        <td><b>' + 书籍号码' + '</b></td>'#13#10;
Result := Result +
        <td><b>' + 书籍名称' + '</b></td>'#13#10;
Result := Result +
        <td><b>' + 出版商' + '</b></td>'#13#10;
Result := Result +
        <td><b>' + 书籍种类' + '</b></td>'#13#10;
Result := Result +
        </tr>'#3#10;
end;
```

现在DoGetQueryResult已经实现完成了，如果再次执行范例 Web应用程序，选择出版商，再选择特定的书籍名称，然后点击“开始查询”按钮，那么就可以在浏览器中看到如图7 11所示的画面。现在我们果然可以正确地查询特定书籍的详细信息了。



图7-11 用书籍的细节信息取代<#QueryResult>定制标记

步骤 5: 使用dbExpress处理Web修改数据

在实现了查询书籍信息的功能之后，再让我们加入能够在线购买书籍的功能，以

此讨论如何在 Web 应用程序中使用 dbExpress 修改数据，而不只是查询数据而已。

首先让我们继续在模板 HTML 文件中加入另外一个 HTML Form 标记，以便输入购买书籍的个人信息。另外在这个 HTML Form 标记中再使用另外一个定制标记 `<#PrepareBooks>`。稍后将用动态产生的 HTML 内容让用户选择想要购买的书籍，并且以此信息取代这个定制标记。再在原先的 HTML Form 中加入一个新的按钮“欲买书籍”。修改后的模板 HTML 文件看起来如图 7-12 所示。



图7-12 修改HTML模板，加入购买书籍的HTML Form标记

修改了模板 HTML 文件之后，接下来需要在范例 Web 应用程序中加入一个新的 WebActionItem，以便在用户选择了想购买的书籍并且输入了个人信息之后，把这些购买信息存储到数据库中。

因此请回到 Delphi 集成开发环境，双击 WebModule，然后像图 7-13 那样加入一个

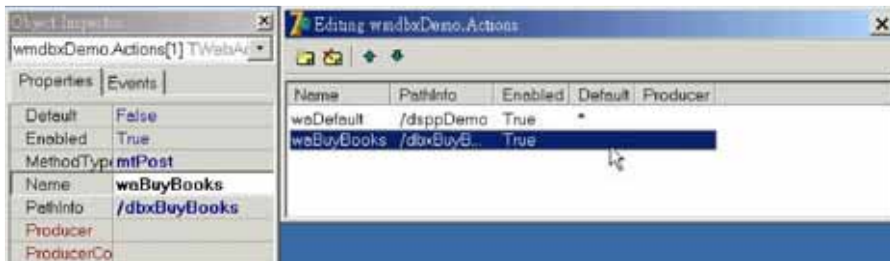


图7-13 在WebModule中加入处理购买书籍的 WebActionItem

新的 WebActionItem，设置它的 Name 特性值为 waBuyBooks，PathInfo 特性值为 /dbxBuyBooks。

现在就可以开始实现购买书籍的功能了。首先修改前面已经说明的 ParseContentFields 函数，判断这个 HTTP 请求是“开始查询”按钮被点击，还是“欲买书籍”按钮被点击。如果是“欲买书籍”按钮被点击，那么就设置全局变量 bPrepareBooks 为 True。

```
procedure TwindbxDemo.ParseContentFields (Request: TWebRequest);
begin
    sPublisher := Request.ContentFields.Values['cbPublisher'];
    sBook := Request.ContentFields.Values['cbBook'];
    if (sBook = '') then
        sBook := NODATAYET;
    sAuthor := Request.ContentFields.Values['cbAuthor'];
    bPrepareBooks := False;
    if (Request.ContentFields.Values['btnPrepareBooks']) <then'
        bPrepareBooks := True;
end;
```

接着修改 ppDefault 的 OnHTMLTag 事件处理函数，加入判断全局变量 bPrepareBooks 的程序代码。如果 bPrepareBooks 是 True，而且目前处理的定制标记是 <#PrepareBooks> 的话，就调用 DoGetPrepareBooks 函数。

```
if (sBook <> NODATAYET) then
    if (bPrepareBooks) then
        begin
            if (TagString = PREPAREBOOKS) then
                begin
                    ReplaceText := DoGetPrepareBooks;
                    Exit;
                end;
        end
    else
        if (TagString = QUERYRESULT) then
            begin
                ReplaceText := DoGetQueryResult;
                Exit;
            end
        else
            ReplaceText := '';
```

最后实现 DoGetPrepareBooks 函数，它和前面介绍的 DoGetQueryResult 函数非常类似，只是它查询特定出版商的所有书籍，并且在组成结果 HTML 表格时在第一个字段中加入了一个 HTML 复选框控件，并且以每一本书的 ISBN 号码加上 ‘ck’ 作为复选框的名称。

```

function TwmDbxDemo.DoGetPrepareBooks: String;
const
    bgColor1 = '#ccccff';
    bgColor2 = '#ffffcc';
var
    sHeader : String;
    bgColor : string;
    sINUT002 : String;
    Row      : Integer;
    aTD: TTransactionDesc;
begin
    Result := '';
    aTD.TransactionID := 1;
    aTD.IsolationLevel := xilREADCOMMITTED;
    scnnDemo.StartTransaction (aTD);

    SetupPrepareBooksQuerySQL;
try
        sHeader := GetPrepareHeader;
        with sdsDemo do
            begin
                Open;
                Result := sHeader + SetupPrepareTableColumn;
                Row := 0;
                while not Eof do
                    begin
                        if (Row mod 2 = 0) then
                            bgColor := bgColor1
                        else
                            bgColor := bgColor2;
                        inc (Row);
                        Result := Result +
                            ' <tr bgcolor="' + bgColor + '">' #13#10;

                        Result := Result +
                            ' <td>' +
                            '<input type="checkbox" name="' + 'ck' + Fields[0].AsString +
                            '" value="ON">' + '</td>' #13#10;
                        Result := Result +
                            ' <td>' + Fields[0].AsString + '</td>' #13#10;
                        Result := Result +
                            ' <td>' + Fields[1].AsString + '</td>' #13#10;
                        Result := Result +

```



```

        <td>' + Fields[2].AsString + '</td>'#13#10;
Result := Result +
        <td>' + Fields[3].AsString + '</td>'#13#10;
Result := Result +
        <td>' + Fields[5].AsString + '</td>'#13#10;

Result := Result +
        </tr>'#13#10;
Next;
end;

Prepared := False;
Close;
end;

Result := Result + '</table>'#13#10;

Result := Result + GetPrepareFooter;
except
    on E: Exception do
    begin
        Result := E.Message;
        scnnDemo.Rollback (aTD) ;
    end;
end;

if scnnDemo.InTransaction then
    scnnDemo.Commit (aTD) ;
end;

procedure TwndbxDemo.SetupPrepareBooksQuerySQL;
var
    sSQL : String;
    sVID : String;
    sAID : String;
begin
    sdsDemo.Active := False;
    sdsDemo.Prepared := False;

    sSQL := sdsDemo.CommandText;
    if (sPublisher <> '') then
    begin
        sVID := GetPublisherVID;
        sSQL := sSQL + ' where VID = ' + '''' + sVID + '''';
    end;
end;

```

```

end;

sdsDemo.CommandText := sSQL;
sdsDemo.Prepared := True;
end;

function TwmdbxDemo.GetPrepareHeader: String;
begin
    Result :=
        '<table border="1" cellpadding="2" cellspacing="2">'#13#10 +
        '  <tr bgcolor="silver">'#13#10;
end;

function TwmdbxDemo.SetupPrepareTableColumn : String;
begin
    Result := Result +
        '    <td><b>' + 购买' + '</b></td>'#13#10;
    Result := Result +
        '    <td><b>' + 'ISBN' + '</b></td>'#13#10;
    Result := Result +
        '    <td><b>' + 书籍编号' + '</b></td>'#13#10;
    Result := Result +
        '    <td><b>' + 书籍号码' + '</b></td>'#13#10;
    Result := Result +
        '    <td><b>' + 书籍名称' + '</b></td>'#13#10;
    Result := Result +
        '    <td><b>' + 书籍种类' + '</b></td>'#13#10;
    Result := Result +
        '  </tr>'#13#10;
end;

function TwmdbxDemo.GetPrepareFooter: String;
begin
    Result := '<p align="center">' +
        '姓名 : <input type="text" name="edtName" size="20">' +
        ' 身份证 : <input type="text" name="edtID" size="20">' +
        '  EMail : <input type="text" name="edtEMail" size="20"> </p>' +
        '<p align="center">' +
        '地址 : <input type="text" name="edtAddress" size="77"> </p>';
end;

```

此外DoGetPrepareBooks函数也动态地产生了用于输入用户个人信息的 HTML单行控件，以便让用户在购买书籍时输入个人信息。

现在编译并且执行范例 Web 应用程序，选择出版商并且点击“欲买书籍”按钮，那么会看到类似于图 7 14 的画面。现在用户可以选择想购买的书籍并且输入个人信息了。现在我们需要完成的最后一个步骤是，在用户点击图 7 14 中的“确定购买”按钮之后，使用 dbExpress 把浏览器中的数据更新回数据库中。

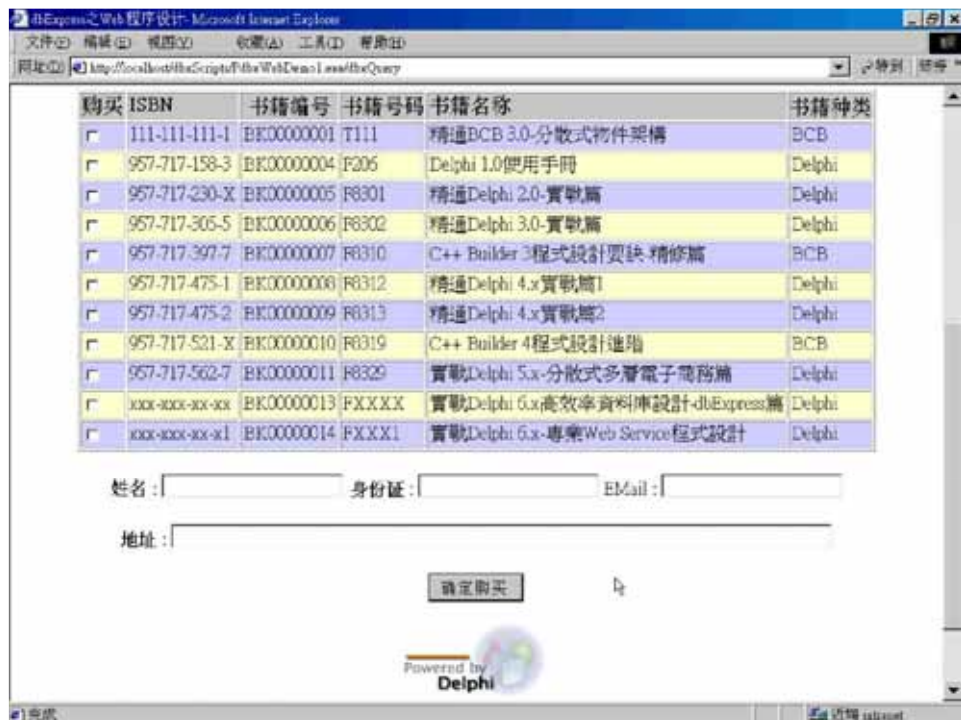


图7-14 在浏览器中查询书籍信息并且准备购买书籍

回到Delphi/Kylix中，并且waBuyBooks的OnAction事件处理函数中编写如下的程序代码：

```
procedure TwmdbxDemo.wmdbxDemo.waBuyBooksAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := DoBuyBooks(Request);
  Handled := True;
end;
```

上面的程序代码在用户点击了图 7 14 中的“确定购买”按钮后会被调用。这个事件处理函数调用 DoBuyBooks 函数来处理用户购买书籍的工作，并且将 TWebRequest 对象传递给 DoBuyBooks。

DoBuyBooks 的实现程序代码如下。它首先调用 GetBuyerInfo 函数从传递进的 TWebRequest 对象中取出用户输入的个人信息，接着它进入一个循环，一一取出用

户选择购买的每一本书，然后调用 **DoBuyBook**，通过使用 **TSQLClientDataSet** 组件把用户购买的书籍的信息连同个人信息更新回数据库中。

```
function TwmdbxDemo.DoBuyBooks (Request: TWebRequest): String;
var
    iCount : Integer;
    sName : String;
    sEMail : String;
    sID : String;
    sAddress : String;
    sISBN : String;
    iPos : Integer;
    aTD: TTransactionDesc;

procedure GetBuyerInfo;
begin
    sName := Request.ContentFields.Values['edtName'];
    sID := Request.ContentFields.Values['edtID'];
    sEMail := Request.ContentFields.Values['edtEMail'];
    sAddress := Request.ContentFields.Values['edtAddress'];
end;

procedure DoBuyBook;
begin
    scdsBuyBooks.Insert;
    scdsBuyBooks.FieldName('ISBN').Value := sISBN;
    scdsBuyBooks.FieldName('BDATE').Value := Now;
    scdsBuyBooks.FieldName('RNAME').Value := sName;
    scdsBuyBooks.FieldName('RID').Value := sID;
    scdsBuyBooks.FieldName('COPIES').Value := 1;
    scdsBuyBooks.FieldName('PMETHOD').Value := 1;
    scdsBuyBooks.FieldName('MAILTO').Value := sAddress;
    scdsBuyBooks.FieldName('EMAIL').Value := sEMail;
    scdsBuyBooks.Post;
    scdsBuyBooks.ApplyUpdates(0);
end;

begin
    GetBuyerInfo;

    aTD.TransactionID := 1;
    aTD.IsolationLevel := xilREADCOMMITTED;
    scnnDemo.StartTransaction(aTD);
try
        scdsBuyBooks.Active True;
```

```

for iCount := 0 to Request.ContentFields.Count do
begin
    sISBN := Request.ContentFields.Names[iCount];
    iPos := Pos('ck', sISBN);
    if (iPos <> 0) then
    begin
        Delete(sISBN, 1, 2); // 删除ck字符
        DoBuyBook;
    end;
end;
Result := '恭喜, 完成购买书籍作业!, 我们会立刻的处理, 谢谢您。';
scdsBuyBooks.Active := False;
except
on e : Exception do
begin
    Result := '错误 : ' + e.Message;
    scnnDemo.Rollback(ATD);
end;
end;
if scnnDemo.InTransaction then
    scnnDemo.Commit(ATD);
end;

```

上面程序代码中使用的 `scdsBuyBooks` 是我们在数据模块中新加入的 `TSQLClientDataSet` 组件, 我们设置它的 `CommandText` 特性值为 `select * from BUYBOOKTRANS`, 再把它的 `PacketRecords` 特性值设置为 1 以便尽可能少地将数据下载到客户端, 因为这个组件的工作主要是把购买书籍的数据更新回数据库中, 我们并不需要使用它访问下载的数据。此时的数据模块如图 7-15 所示。



图7-15 在数据模块中加入 `TSQLClientDataSet` 组件 `scdsBuyBooks` 来处理用户购买书籍的信息

现在我们就可以在浏览器中执行这个实现了修改数据功能的范例 Web 应用程序了。在浏览器中选择要购买的书籍，那当然是 Delphi 的相关书籍，然后输入个人信息，最后再点击“确定购买”按钮（见图 7 16）。



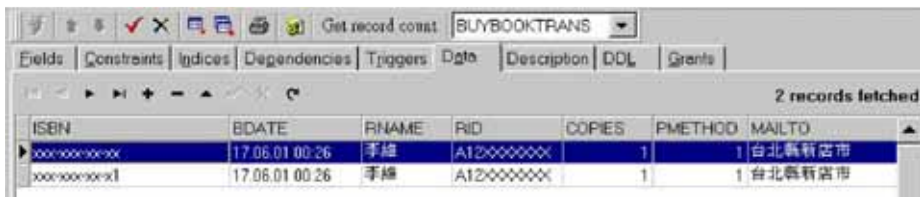
图7-16 在浏览器中选择欲购买的书籍并且输入个人信息

在点击了“确定购买”按钮而且范例 Web 应用程序成功地更新了数据之后，它会显示如图 7 17所示的画面，代表范例 Web 应用程序成功地更新了数据。



图7-17 Web应用程序给出成功购买的消息

在看到图 7 17所示的画面之后，如果我们直接到 InterBase 中查看数据，那么就可以看到类似于图 7 18的画面，范例 Web 应用程序果然成功地通过 dbExpress 将数据更新到了数据库中。



The screenshot shows a database browser window with a table named 'BUYBOOKTRANS'. The table has columns: ISBN, BDATE, RNAME, RID, COPIES, PMETHOD, and MAILTO. Two records are displayed, both with ISBN 'xxxxxxxxxx', BDATE '17.06.01 00:26', RNAME '李雄', RID 'A12xxxxxxxxx', COPIES '1', PMETHOD '1', and MAILTO '台北縣新店市'.

ISBN	BDATE	RNAME	RID	COPIES	PMETHOD	MAILTO
xxxxxxxxxx	17.06.01 00:26	李雄	A12xxxxxxxxx	1	1	台北縣新店市
xxxxxxxxxx	17.06.01 00:26	李雄	A12xxxxxxxxx	1	1	台北縣新店市

图7-18 在浏览器中选择的书籍信息真的更新到数据库中了

好了，上面的步骤详细说明了如何在 Web应用程序中使用 dbExpress处理数据。从这些步骤中可以看到使用 dbExpress非常简单，但是它执行起来却非常有效率，而且极为节省资源，可以让 Web应用程序拥有更大的延展性。

步骤 6: 转变成ISAPI

如果读者使用 Kylix，那么可以跳过本小节讨论的内容。

前面开发的范例都是 CGI类型的Web应用程序，但是对于许多人来说，为了性能和延展性的因素，可能需要开发 ISAPI类型的Web应用程序，或是开发 Apache中 Shared Module类型的Web应用程序。Delphi允许程序员轻易地将 CGI类型的Web应用程序转换为 DLL类型的Web应用程序，以提供更好的性能。本小节讨论的内容就在于告诉读者如何将 CGI的范例Web应用程序转换为 ISAPI类型的Web应用程序。

首先在 Delphi/Kylix中建立一个新的项目组，然后加入前面开发的 CGI项目。然后在此项目组中建立一个新的 Web应用程序项目，选择建立 ISAPI类型的Web应用程序（见图 7 19）。

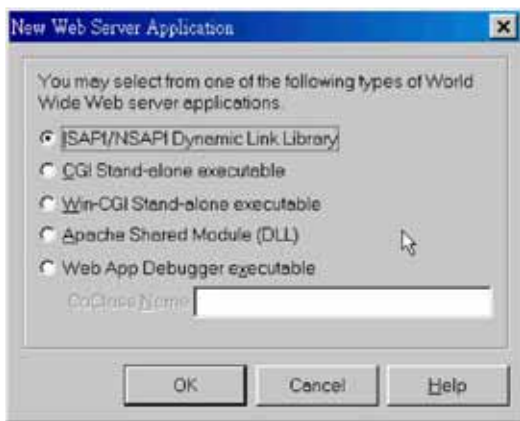


图7-19 选择建立ISAPI类型的Web应用程序

然后回到项目组中，删除刚才建立的 ISAPI应用程序自动建立的空白 WebModule，再加入原先在 CGI应用程序中的 WebModule，然后存储此 ISAPI应用程序，至此便成

成功地建立了ISAPI应用程序。图7 20是此时项目组中 CGI和ISAPI应用程序的项目，请注意这两个项目都使用了相同的 WebModule，uwmdbxWebDemo1。这个WebModule是原先在CGI应用程序中开发的。

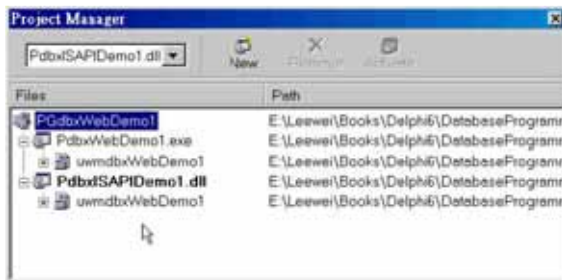


图7-20 在新项目中加入原先在CGI应用程序中的WebModule

我们希望这个WebModule能够同时使用在CGI和ISAPI应用程序中，但是在前面开发CGI类型的应用程序时，本书使用的HTML模板在HTML的form标记部分使用了如下的程序代码：

```
<form method="POST" action="/dbxScripts/PdbxWebDemo1.exe/dbxQuery">
...
```

请注意，在这行程序代码中我们直接使用了 CGI应用程序的名称PdbxWebDemo1.exe作为处理此Post行动的程序。同样，当我们在购买书籍时，也是直接使用 CGI应用程序的名称：

```
<form method="POST"
action="/dbxScripts/PdbxWebDemo1.exe/dbxBuyBooks">
..
```

但是在现在要开发的 ISAPI应用程序中，却需要使用 ISAPI应用程序的名称，如下所示：

```
<form method="POST"
action="/dbxScripts/PdbxISAPIDemo1.DLL/dbxBuyBooks">
..
```

要解决这个问题很简单，我们只需要把 Web应用程序的名称变成一个定制标记，然后在Web应用程序执行时获取正确的 Web应用程序名称，然后动态地替换此定制标记即可。因此我们可以先修改此 Web应用程序使用的HTML模板，把上述拥有Web应用程序名称的部分改为定制标记 <#WebAppName>。例如把刚才的HTML程序代码改为：

```
<form method="POST" action="/dbxScripts/<#WebAppName>/dbxBuyBooks">
..
```

然后在原先的 Web应用程序中加入替换此定制标记的程序代码。我们修改原先WebModule中ppDefault组件的 OnHTMLTag事件处理函数，加入如下处理

<#WebAppName>的程序代码:

```
if (TagString = APPNAME) then
begin
  ReplaceText := sAppName;
  exit;
end;
```

上面的程序代码用 sAppName 取代 <#WebAppName> 定制标记。sAppName 则是一个全局变量。在 WebModule 的 OnCreate 事件处理函数中调用 GetModuleFileName 来取得 Web 应用程序的正确名称并且将它存储在 sAppName 中:

```
procedure TwndbxDemo.WebModuleCreate (Sender: TObject);
var
  cBuffer : String[255];
  iLen : Integer;
begin
  iLen := GetModuleFileName(TInstance, @cBuffer, size(cBuffer));
  sAppName := Copy(cBuffer, 1, iLen);
  sAppName := ExtractFileName(sAppName);
end;
```

经过了这些修改之后, 就可以编译 PdbxISAPIDemo1 项目, 这会产生一个 PdbxISAPIDemo1.DLL, 接着把这个 DLL 复制到 Web 的虚拟目录中, 再执行此 DLL, 那么读者就可以看到和前面 CGI 应用程序功能相同的 ISAPI Web 应用程序, 但是执行速度却快得多。通过 Delphi 的 Web 向导, Delphi 的程序员可以在开发时选择建立方便的 CGI 应用程序, 在开发完成之后, 再选择建立执行速度良好的 ISAPI 应用程序。

现在, 如果你执行 ISAPI 类型的范例 Web 应用程序, 你会发现 ISAPI 类型的 Web 应用程序比 CGI 类型的 Web 应用程序快了至少一倍, 而且需要使用的资源比 CGI 更少。

7.3 dbExpress 和 ntraWeb

如果读者使用 Kylix, 那么请自行到 IntraWeb 的网站下载 IntraWeb For Kylix, 安装之后再继续阅读本节。读者可在下列的 URL 找到 IntraWeb For Kylix:

<http://www.atozedsoftware.com/intraweb/download/FilesRelease.html>

Borland 在 Delphi 7 中引入了 Atozedsoftware 公司的 IntraWeb, 让 Delphi/Kylix 的开发 者终于能够使用 RAD 的方式来开发 Web 应用系统。IntraWeb 不但提供了以可视化拖曳方式设计 Web 应用系统的能力, 而且能够使用 Delphi/Kylix 的数据访问引擎开发各种类型的 Web 应用程序。IntraWeb 提供的功能比目前 Microsoft.NET 的 WebForm 还强大, 是 Delphi/Kylix 程序员在开发 Web 应用系统时的帮手。

由于 IntraWeb 提供了大量而且强劲的功能，因此笔者建议有兴趣的读者参考 IntraWeb 的相关文件。目前 IntraWeb 除了提供 Delphi/Kylix/C++Builder 版之外，也计划提供 Java 以及 .NET 的 IntraWeb 版本，读者可以参考下面的 URL 得到进一步的信息：

<http://www.atozedsoftware.com/intraweb.html>

由于 IntraWeb 提供了丰富的 Web 功能，而 dbExpress 则是非常有效率的数据访问引擎，因此结合 IntraWeb 和 dbExpress 将可提供很好的 Web 技术解决方案。因此本小节讨论的内容就是展示如何使用 IntraWeb 和 dbExpress 来开发 Web 应用程序。由于本书不是专门讨论 IntraWeb 的书籍，因此笔者建议读者在阅读完本章之后再去阅读更多有关 IntraWeb 的资料。

接下来的范例将使用 IntraWeb 和 dbExpress 开发一个访问数据库的 Web 应用程序。在这个范例中除了展示如何集成 IntraWeb 和 dbExpress 之外，也将展示 IntraWeb 动态建立控件的功能。这个 Web 应用程序允许用户使用浏览器浏览数据库中的数据表，并且浏览任一数据表中的数据，接着范例 Web 应用程序将使用动态建立 IntraWeb 控件的能力来让用户修改数据表中的数据。

建立 IntraWeb 应用程序

点击 File New 菜单，在 IntraWeb 选项卡中选择建立包含数据模块的 IntraWeb 应用程序，如图 7-21 所示。



图7-21 建立 IntraWeb 应用程序

点击 OK 按钮之后，IntraWeb 会要求程序员将 IntraWeb 项目存储在特定的目录下。接着在项目中的数据模块中放入 TSQLConnection 和 TSQLDataSet 组件。将 TSQLConnection 连接到范例 InterBase 数据库 D7Books，此时数据模块应该看起来如图 7-22 所示。

现在我们就可以在 IntraWeb 项目的主 WebApp 窗体中使用拖曳的方式通过

IntraWeb控件设计Web应用程序的图形用户界面。首先开启 WebApp窗体，在左边放入一个TIWListbox控件用于显示数据库中的所有数据表名称，在上方放入一个 TIWDBNavigator用于浏览数据表的数据。再放入三个 TIWButton控件，分别用于取得数据表信息、取得特定数据表的信息以及允许用户编辑数据表中的数据。最后在 WebApp窗体上方放入 TIWEdit、TIWLabel和一个TDataSource控件，再把TDataSource的DataSet特性值连接到数据模块中的 TSQLDataSet sdsGeneral上。此时 WebApp窗体如图7 23所示。

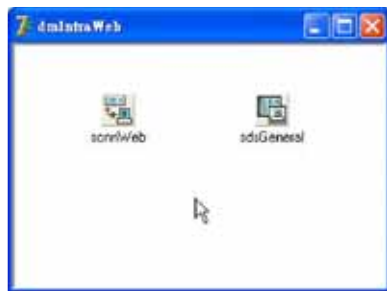


图7-22 在IntraWeb建立的数据模块中放入dbExpress组件

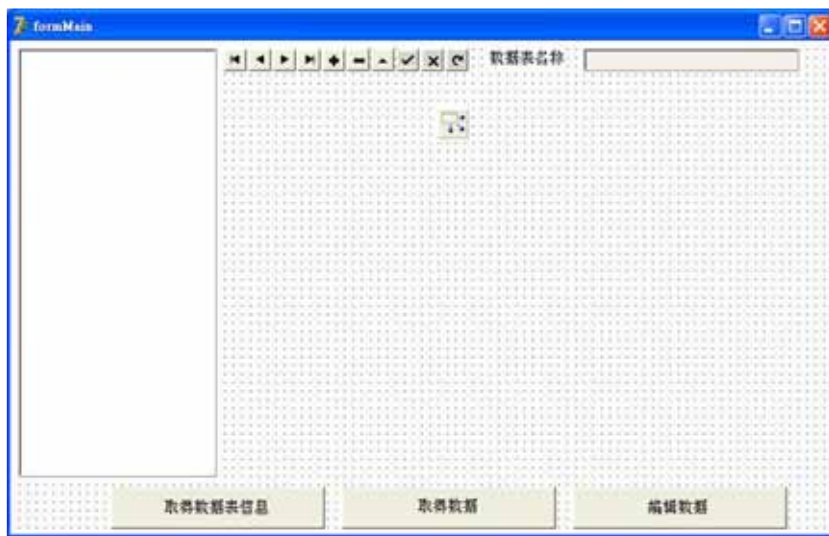


图7-23 范例IntraWeb的主WebApp窗体

现在先让我们实现图7 23中“取得数据表信息”按钮的功能。这个按钮能够从数据库中取得数据表名称。下面是它的实现程序代码：

```
procedure TFormMain.iwbtnGetTablesClick(Sender: TObject);
var
  atblList : TStringList;
begin
  Self.IWListbox1.Items.Clear;
  atblList := TStringList.Create;
  try
    dmIntraWeb.scnnWeb.GetTableNames(atblList, False;
    Self.IWListbox1.Items.Assign(atblList);
```

```
finally  
    FreeAndNil (atblList);  
end;  
end;
```

上面的程序代码首先建立一个 **TStringList**对象，使用它调用 **TSQLConnection**的 **GetTableNames**方法以取得数据库中的数据表名称，再指定给 **WebApp**窗体中的 **IWListbox1**。从这些程序代码中读者会发现，使用 **IntraWeb**开发Web应用程序就像开发一般的Delphi/Kylix应用程序一样，使用 **Object Pascal**就可以完成许多工作。这就是使用 **IntraWeb**的好处，**IntraWeb**让程序员既可以以可视化方式设计Web图形用户界面，又可使用熟悉的 **Object Pascal**来编写应用程序的逻辑程序代码。

接着让我们再实现“取得数据”按钮的功能。当用户点击了这个按钮之后，范例Web应用程序就从 **IWListbox1**中取得用户选择的数据表名称，然后到后端数据库中取得这个数据表的所有数据，再通过 **IntraWeb**的 **TIWDBGrid**控件呈现在浏览器中。由于每一个数据表的结构都不一样，而 **TIWDBGrid**控件会在其中集成特定数据表的字段信息，因此我们无法直接在 **WebApp**窗体中放入 **TIWDBGrid**控件，我们需要动态地建立 **TIWDBGrid**控件。下面的实现程序代码首先调用 **ReCreateDBGrid**动态建立 **TIWDBGrid**，再组合SQL语句并且执行它来访问用户选择的数据表的数据。

```
procedure TformMain.iwbtnGetDataClick (Sender: TObject;  
begin  
    ReCreateDBGrid;  
    dmIntraWeb.sdsGeneral.Active := False;  
    dmIntraWeb.sdsGeneral.DataSet.CommandText :=  
        'Select * from ' + IWListbox1.Items[IWListbox1.ItemIndex];  
    dmIntraWeb.sdsGeneral.Active := True;  
end;
```

ReCreateDBGrid除了建立 **TIWDBGrid**之外，也需要设置它的许多特性值才能正确地显示在浏览器中。在下面的程序代码中直接使用常数值设置特性值，读者可以根据应用程序的设计动态地计算这些特性值。最后，**ReCreateDBGrid**指定了 **TIWDBGrid**控件的 **OnRenderCell**事件处理函数，以便在显示数据时使用特别的颜色来呈现数据。

```
procedure TformMain.ReCreateDBGrid;  
begin  
    if Assigned (aDBGrid) then  
        FreeAndNil (aDBGrid);  
    aDBGrid := TIWDBGrid.Crea(Self);  
    aDBGrid.Left := 200;  
    aDBGrid.Top := 32;
```

```

aDBGrid.Width := 561;
aDBGrid.Height := 377;
aDBGrid.OnRenderCell := IWDBGrid1RenderCell;
aDBGrid.Parent := Self;
aDBGrid.DataSource := Self.dsGeneral;
end;

```

IWDBGrid1RenderCell事件处理函数实现了下面的程序代码，除了网格表头使用不同的背景颜色之外，目前数据的背景颜色使用黄色。

```

procedure TFormMain.IWDBGrid1RenderCell (ACell: TIWGridCell; const ARow,
    AColumn: Integer;
begin
    if ARow = 0 then
    begin
        ACell.BGColor := clSilver;
        if AColumn = 0 then
        begin
            ACell.Font.Color := clRed;
        end;
    end
    else
    begin
        if Assigned(aDBGrid) then
            if aDBGrid.RowIsCurrent then
            begin
                ACell.BGColor := clYellow;
            end;
        end;
    end;
end;

```

另外，当用户在 TIWListbox中选择不同的数据表名称时， TIWListbox的 OnChange事件处理函数也会把目前选择的数据表名称显示在 WebApp主窗体的 TIWListbox中。

```

procedure TFormMain.IWListbox1Change (Sender: TObject;
begin
    iwedtTableName.Caption := IWListbox1.Items[IWListbox1.ItemIndex];
end;

```

接着让我们再建立另外一个 WebApp窗体以便让用户可以编辑数据表的数据。请点击图7 21中的 Application Form图标建立新的 WebApp窗体。然后在这个 WebApp窗体中放入如图7 24所示的控件。在这个新的 WebApp窗体中，范例 IntraWeb应用程序将根据用户选择的数据表的结构动态建立 IntraWeb控件来显示和编辑数据。

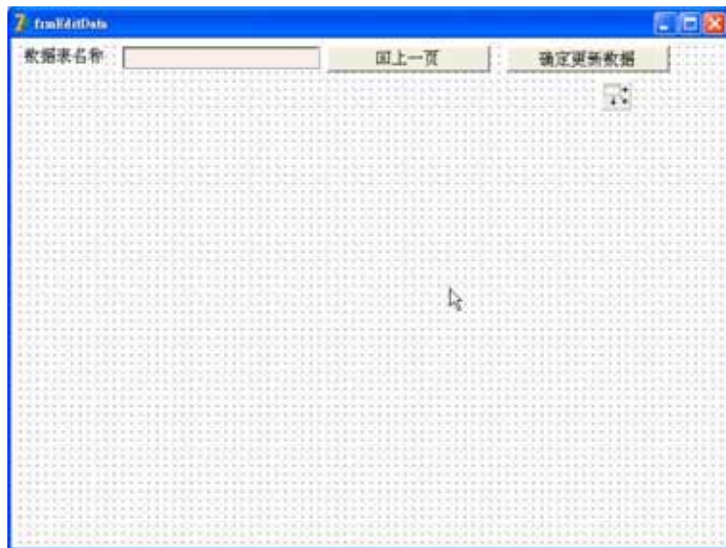


图7-24 范例IntraWeb应用程序中可以修改数据的 WebApp窗体

首先我们实现 WebApp 主窗体中的“编辑数据”按钮的事件处理函数，它先建立图7 24中的 WebApp 窗体，接着调用这个 WebApp 窗体中的两个公有方法以设置用户选择的数据表名称，再要求它根据选择的数据表动态建立 IntraWeb 控件，最后调用 Show 在浏览器中显示第二个 WebApp 窗体。

```
procedure TformMain.iwbtnEditDataClick (Sender: TObject;
var
    aEditForm : TfrmEditData;
begin
    aEditForm := TfrmEditData.CreateWebApplication;
    aEditForm.SetTableName (IWListbox1.Items[IWListbox1.ItemIndex]);
    aEditForm.CreateWebControls;
    aEditForm.Show;
end;
```

下面是第二个 WebApp 窗体的实现程序代码。其中最主要的方法是 CreateWebControls。CreateWebControls 根据数据模块中 sdsGenerasl 连接的数据表（也就是用户选择的数据表）的字段信息逐一建立 IntraWeb 的 TIWLabel 和 TIWDBEdit 控件以显示每一个字段的数据。CreateWebControls 基本上是一个范例，如果在实际的应用中，读者应该根据每一个字段的类型建立不同的 IntraWeb 控件来正确地显示数据。

```
const
    ITOP = 60;
    IDELTA = 26;
    ILEFT1 = 16;
```



```

ILEFT2 = 160;
IWIDTH = 400;

procedure TfrmEditData.CreateWebControls;
var
    iTopPos : Integer;
    iCount : Integer;
    aWebLabel : TIWLabel;
    aWebControl : TIWDBEdit;
    aField : TField;

procedure CreateWebLabel;
begin
    aWebLabel := TIWLabel.Create(Self);
    aWebLabel.Top := iTopPos;
    aWebLabel.Left := ILEFT1;
    aWebLabel.Caption := aField.FieldName;
    aWebLabel.Parent := Self;
end;

procedure CreateWebControl;
begin
    aWebControl := TIWDBEdit.Create(Self);
    aWebControl.Top := iTopPos;
    aWebControl.Left := ILEFT2;
    aWebControl.DataSource := Self.dsGeneral;
    aWebControl.DataField := aField.FieldName;
    aWebControl.Parent := Self;
    aWebControl.Width := IWIDTH;
end;

begin
    iTopPos := ITOP;
    for iCount := 0 to dmIntraWeb.sdsGeneral.FieldCount do 1
    begin
        aField := dmIntraWeb.sdsGeneral.Fields[iCount];
        CreateWebLabel;
        CreateWebControl;
        iTopPos := iTopPos + IDELTA;
    end;
end;

procedure TfrmEditData.iwbtnGoBackClick (Sender: TObject;

```

```

begin
    Self.Release;
end;

procedure TfrmEditData.SetTableName (const sName: String)
begin
    Self.iwedtTableName.Caption := sName;
end;

procedure TfrmEditData.iwbtnPostClick (Sender: TObject;
begin
    dmIntraWeb.sdsGeneral.Post;
    dmIntraWeb.sdsGeneral.ApplyUpdates (0);
end;

```

第二个WebApp窗体中的“回上一页”按钮只需要调用Self.Release就可以释放这个Web页并且回到主WebApp窗体中。“确定更新数据”按钮调用数据模块中sdsGeneral的ApplyUpdates，把用户在这个WebApp窗体中修改的数据更新回后端的数据表中。

现在编译并且执行此范例 IntraWeb应用程序，那么我们会先看到图 7 25所示的 IntraWeb Application Server被激活。接着我们点击Launch Program图标激活IE并且执行范例IntraWeb应用程序。

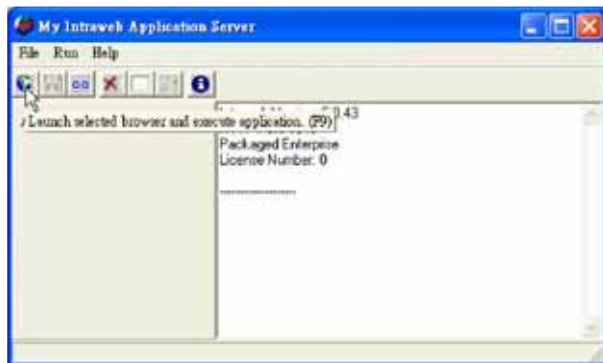


图7-25 执行范例IntraWeb应用程序时会先激活 IntraWeb Application Server

在IE被激活并且加载范例 IntraWeb应用程序之后，我们可以点击其中的“取得数据表信息”按钮，那么就可以看到类似于图 7 26的画面，D7Books数据库中的所有数据表名称被显示在TIWListbox中。

接着，在TIWListbox中选择一个数据表名称，再点击浏览器中的“取得数据”按钮，那么就可以在浏览器中看到 TIWDBGrid控件出现并且呈现了数据表中的所有数据，如图7 27所示。

最后点击“编辑数据”按钮，那么范例 IntraWeb应用程序便会显示第二个 Web

App窗体，并且动态建立 IntraWeb控件显示数据并且允许用户修改数据，如图 7 28所示。最后用户可以点击图 7 28中的“确定更新数据”按钮把修改过的数据更新回数据表中，或是点击“回上一页”按钮重新回到图 7 27的画面。

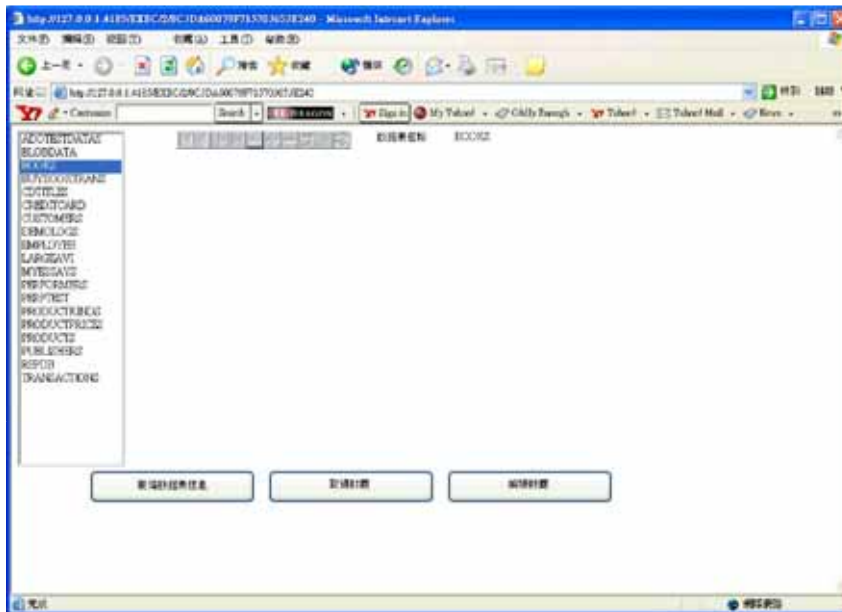


图7-26 范例IntraWeb应用程序取得数据表的名称



图7-27 范例IntraWeb应用程序取得Books数据表中的数据

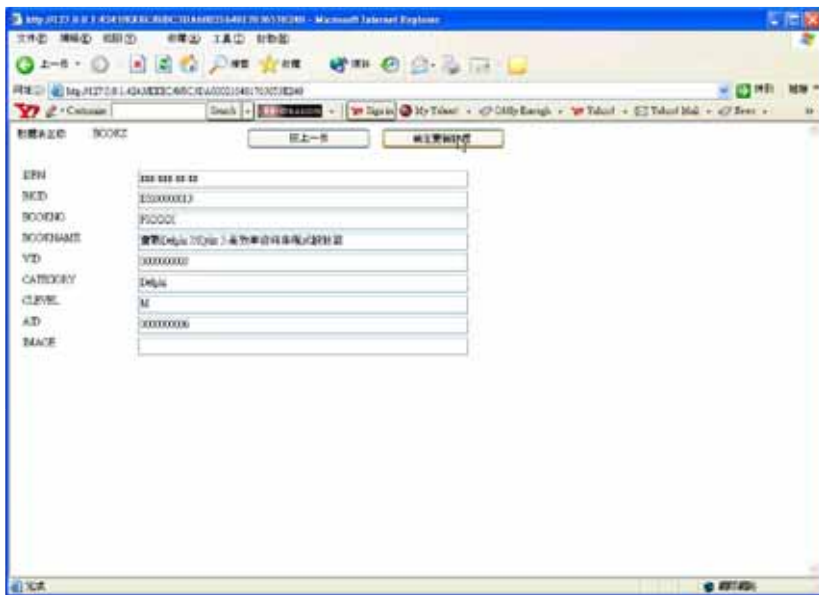


图7-28 范例IntraWeb应用程序显示可以修改数据的画面

本小节的IntraWeb范例展示了如何结合使用IntraWeb控件和dbExpress以及动态建立IntraWeb控件的功能。读者现在应该知道IntraWeb结合dbExpress可以发挥出很大的能力。事实上IntraWeb也允许程序员结合WebBroker的功能来开发更有弹性的Web应用系统。

本小节展示的IntraWeb应用程序并没有考虑到美工和图形用户界面方面的设计，读者可以自己开发一个更具专业水准的IntraWeb应用程序或是修改此范例IntraWeb应用程序。例如为范例IntraWeb应用程序加入底图的模板等，这些就留给读者来完成了。

7.4 结论

本章讨论了如何使用dbExpress来开发Web应用系统，通过结合Delphi的WebBroker功能和dbExpress快速访问数据的能力，Delphi/Kylix的程序员能够使用这两种技术开发最具性能的Web解决方案。

由于dbExpress具有单向、只读的数据访问能力，这正好是大部分Web应用程序访问数据的特点，因此dbExpress和WebBroker几乎是最佳的绝配。由于从Delphi 7之后WebBroker和dbExpress都能够同时在Windows和Linux平台上执行，因此对于需要在这两个平台上开发Web应用系统的软件人员来说，WebBroker和dbExpress具有极强的吸引力。如果程序员再能够使用Delphi 7和Kylix提供的WebSnap功能来开发功能强劲、特性丰富的Web应用系统，那么几乎没有任何其他工具能够提供更好的Web

开发技术了。

在本章的最后说明了如何使用 Delphi 7 最新提供的 IntraWeb 以可视化的方式开发 Web 应用系统。由于 IntraWeb 提供了以往 Delphi/Kylix 在 Web 方面最弱的 GUI 设计功能，同时 IntraWeb 又提供了 Windows/Linux/.NET 的版本以及 Java 的版本，因此已经成为一个跨平台的强大 Web 解决方案。更重要的是，IntraWeb 可以与 dbExpress 一起合作开发最具性能的 Web 数据库应用程序，Delphi/Kylix 加上 IntraWeb 无疑是现在 Windows/Linux 和未来 .NET 上最强大的 Web 应用方案之一。

第三部分

dbExpress高级技术篇



第8章 处理二进制大型数据

在许多应用系统中，除了要处理一般类型的数据，例如文本数据、数值数据等之外，也有许多应用程序需要处理二进制的的数据，例如图形数据、声音数据、大量文本数据等。这种二进制的数据类型一般称为二进制大型数据（BLOB）。

Delphi 7的dbExpress完整支持了开发BLOB类型的应用程序，程序员不但可以使用dbExpress处理图形数据或是处理声音数据，也可以使用 dbExpress处理动画数据。dbExpress中的BLOB字段对象让程序员可以非常方便地处理 BLOB类型的数据。

本章讨论的内容就是说明如何使用 dbExpress来处理BLOB类型的数据，本章先说明如何使用dbExpress开发一般的处理图形数据的应用程序，让读者掌握 dbExpress处理BLOB类型数据的基本技巧。

但是如果读者需要开发处理大量 BLOB类型的数据，就必须学习如何有效率地使用dbExpress来处理BLOB数据，如此一来才能够编写出性能良好的 BLOB应用程序。但是如何才能使用 dbExpress适当地处理大量的 BLOB数据呢？这就是本章要讨论的重点之一。在了解了如何善用 dbExpress处理大量的BLOB数据后，读者可以有信心地开发BLOB类型的应用系统。

此外在 Windows平台中，Microsoft提供了OLE Container的功能，使程序员能够使用这项功能巧妙地处理 BLOB类型的数据。例如激活 Excel编辑XLS类型的数据，或是激活计算机中处理图形的程序。本章也将说明如何结合 dbExpress处理BLOB类型数据的能力和OLE Container的功能。

8.1 处理图形数据

首先让我们使用一个简单的范例来说明如何使用 dbExpress处理图形数据。这个范例让读者能够输入一些数据，在某一个字段中加载图形数据，并且能够在数据感知组件中显示图形。在下面的步骤中将逐步完成这个图形数据范例。

下面便是开发步骤：

- 1) 在Delphi集成开发环境中点击File New Application菜单建立一个新的项目。
- 2) 在主窗体中放入TDBNavigator、TDBGrid、TDBBitmap、TDataSource和三个TBitBtn按钮。如图8.1所示。
- 3) 设置TDBNavigatorTDBGridTDBBitmap的DataSource特性值为主窗体中的TDataSource组件。

4) 最后放入一个位于 Dialogs选项卡中的TOpenDialog组件。



图8-1 范例程序的主窗体

5) 点击File New DataModule菜单以建立一个数据模块。

6) 在数据模块中放入 TSQLConnection组件并且连接到数据库，这个范例连接到本书光盘中的 Delphi7DB.gdb范例数据库，设置 Connected特性值为 True。

7) 放入一个 TSimpleDataSet组件，设置它的 DBConnection特性值为步骤6中放入的 TSQLConnection组件，再设置它的 DataSet\CommandText特性值为 select * from PRODUCTS。设置 Active特性值为 True。

8) 再放入一个 TSQLDataSet，设置它的 SQLConnection特性值为步骤6中放入的 TSQLConnection组件，再设置它的 CommandText 特性值为 select Max (PID) from PRODUCTS。

现在范例数据模块看起来如图 8 2所示。

9) 双击 TSimpleDataSet组件，并且加入代表所有字段的字段对象（见图 8 3）。

10) 回到主窗体，设置它的 TDBImage组件的 DataField特性值为 PIMAGE。

11) 回到主窗体，点击File Use Unit...加入步骤5建立的数据模块。

现在就可以执行这个范例并且在 TDBGrid中将数据输入到数据库中。但是在执行之前，让我们在主窗体中的“载入影像”按钮中开启对话框让用户开启一个图形文件，并且存储在 PRODUCTS数据表的 PIMAGE 字段中。PIMAGE 字段是 BLOB 类型的 InterBase 字段。



图8-2 建立数据模块并且放入 dbExpress组件

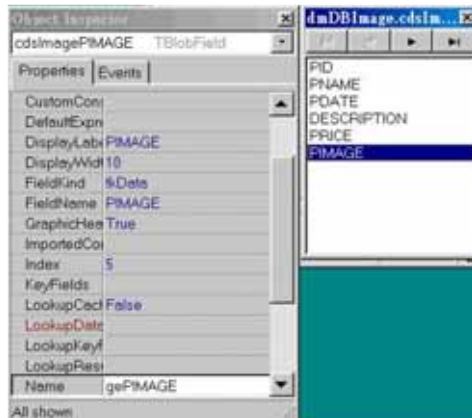


图8-3 激活TSimpleDataSet的字段编辑器，加入所有字段对象

12) 点击主窗体中的“载入影像”按钮，并且在OnClick事件处理函数中编写如下程序代码：

```
procedure TForm1.bbtnLoadImageClick(Sender: TObject);
begin
    if opDlg.Execute then
    begin
        if (dmDBImage.cdsImage.State = dsBrowse) then
            dmDBImage.cdsImage.Edit;
        dmDBImage.cdsImagePIMAGE.LoadFromFile(opDlg.FileName);
    end;
end;
```

上面的程序代码首先调用主窗体中 TOpenDialog 组件的 Execute 方法，以便让用户加载一个 BMP 类型的文件。如果用户指定了 BMP 文件，那么 Execute 会返回 True，于是范例应用程序再检查目前的数据表是否是处于编辑状态，如果不是的话，就调用 TSimpleDataSet 的 Edit 方法以进入编辑数据的状态。

接着范例应用程序调用了代表 PIMAGE 的字段对象 cdsImagePIMAGE 的 LoadFromFile。cdsImagePIMAGE 的类型是 TBlobField，它代表存储 BLOB 类型数据的字段，而它的 LoadFromFile 方法则可以从一个文件中加载文件的内容。由于此时加载的是存储 BMP 图形的文件，因此这个 BMP 文件的内容将存储到 cdsImagePIMAGE 代表的数据表字段中。

13) 点击主窗体中的 ApplyUpdates 按钮，并且在事件处理函数中调用 TSimpleDataSet 的 ApplyUpdates 方法把用户输入的数据更新回后端的数据库中。

```
procedure TForm1.bbtnApplyUpdatesClick(Sender: TObject);
begin
    dmDBImage.cdsImage.ApplyUpdates(0);
end;
```

现在我们就可以执行这个范例应用程序了，图 8 4是在范例应用程序中输入数据、加载BMP文件和将数据更新回数据表的连续画面，从画面中可以看到 dbExpress果然能够正确地处理图形种类的数据。

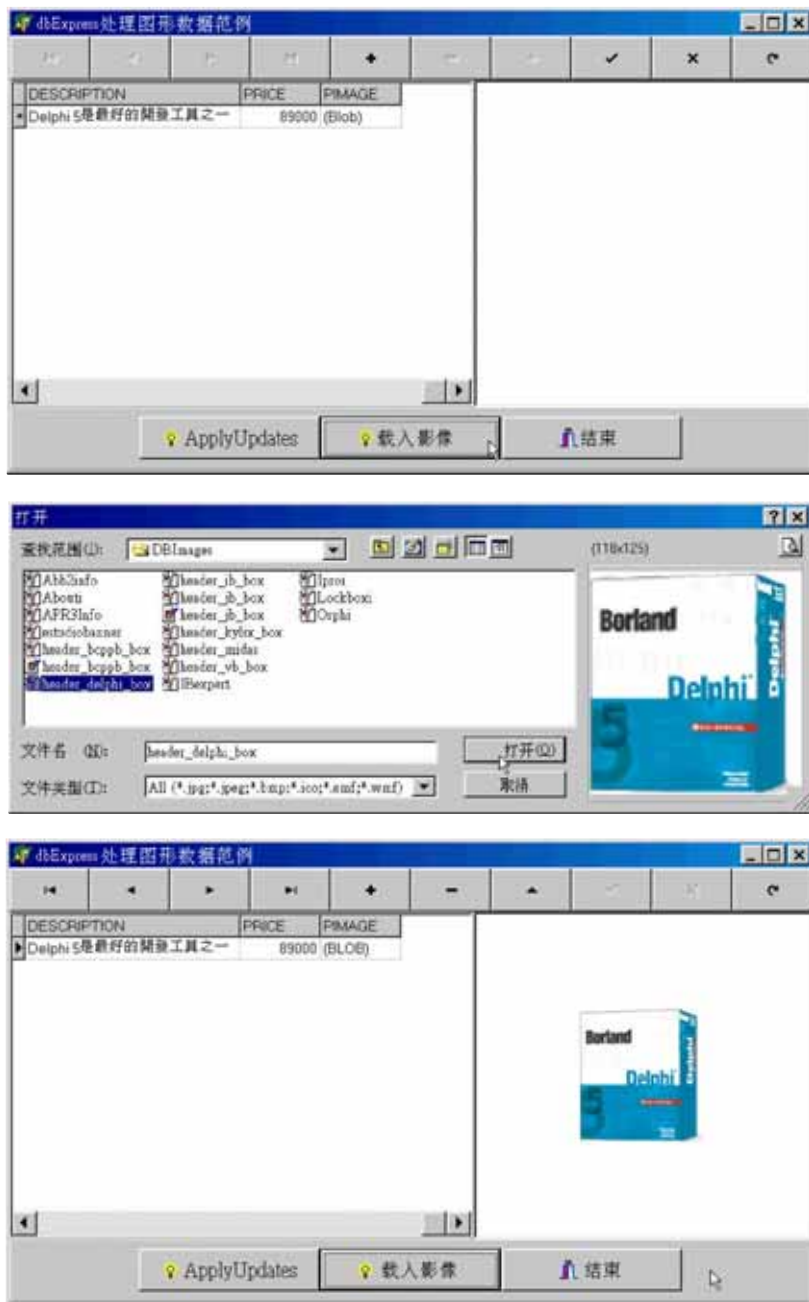


图8-4 在范例应用程序中输入数据、加载BMP文件和将数据更新回数据表的连续画面

图8 5则是使用范例应用程序输入多个图形数据之后的画面。从图 8 5中我们可以知道, dbExpress在处理图形数据时就和处理一般的文本数据一样, 相当方便, 只要程序员了解了如何使用 dbExpress的TBlobField类型的字段对象, 就可以轻松地处理BMP类型的图形数据。



图8-5 范例应用程序输入了多个图形数据之后的画面

8.2 处理JPEG类型的图形数据

在上一小节中处理的数据是属于 BMP类型的图形数据, 但是对于许多应用来说, BMP文件实在是太大了, 为了有效地使用资源, 一般的应用程序都会使用压缩的图形数据。Delphi 7允许程序员处理 JPEG类型的数据, 并且能够把 JPEG类型的数据存储在BLOB类型的字段中。本小节讨论的内容就是展示如何使用 dbExpress处理JPEG类型的图形数据。

1) 像上一小节一样点击 File New Application菜单建立一个新的项目, 并且建立相同的主窗体。但是把 TDBImage改成 TImage, 并且设置它的Name特性值为 igJPG如图8 6所示。

2) 在主窗体的uses句子中加入jpeg程序单元, 以便处理 JPEG格式的图形。

uses

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, DB, Grids, DBGrids, ExtCtrls, DBCtrls, StdCtrls, Buttons, ExtDlgs,  
jpeg;
```

3) 按照上一小节的步骤 12和步骤 13为这个范例的主窗体中的“载入影像”按钮

以及“ApplyUpdates”按钮编写和上一小节相同的程序代码。

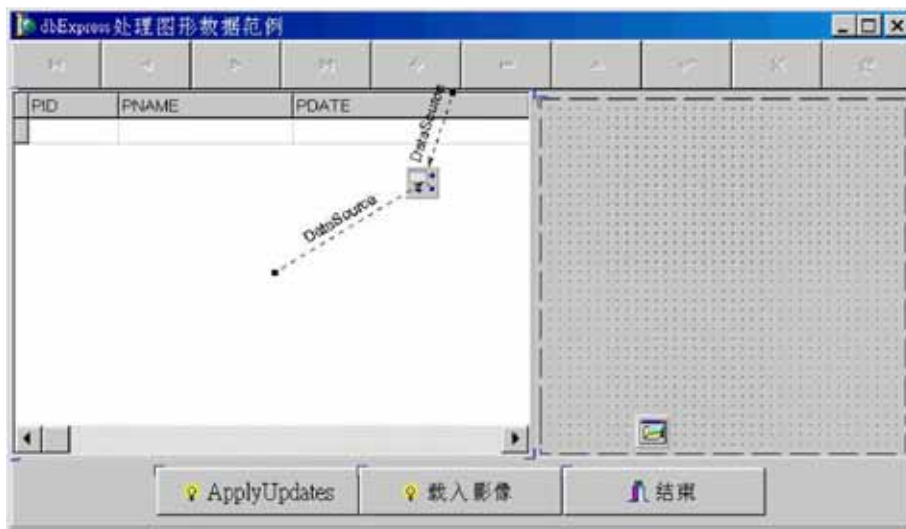


图8-6 在主窗体中加入TImage组件

4) 现在在数据表的PIMAGE字段中存储的是JPEG类型的图形数据，而TDBImage组件无法直接显示JPEG格式的图形，因此本范例使用了TImage组件。为了让TImage能够显示PIMAGE字段中的JPEG图形，回到数据模块程序单元中，在TSimpleDataSet组件的AfterScroll事件处理函数中编写如下的程序代码：

```
procedure TdmDBImage.cdsImageAfterScroll (DataSet: TDataSet);
begin
    try
        cdsImagePIMAGE.SaveToFile ('Temp.JPG') ;
        Form1.igJPG.Picture.LoadFromFile ('Temp.JPG') ;
    except
        on Exception do;
    end;
end;
```

在上面的程序代码中，首先调用代表PIMAGE字段的字段对象cdsImagePIMAGE的SaveToFile方法把字段中的JPEG数据存储到一个暂时的Temp.JPG中，再调用TImage组件igJPG的LoadFromFile方法从刚才的Temp.JPG文件中加载JPEG图形数据，并且显示出来（见图8-7）。由于在前面第二个步骤中范例应用程序加入了jpg程序单元，因此TImage组件可以处理JPEG类型的数据。

现在执行本范例应用程序，在TDBGrid中输入一些数据，再点击“载入影像”按钮以加载一个JPEG类型的图形文件，便可以知道这个范例应用程序现在可以正确地处理JPEG类型的图形数据了。图8-7便是范例应用程序执行的画面。

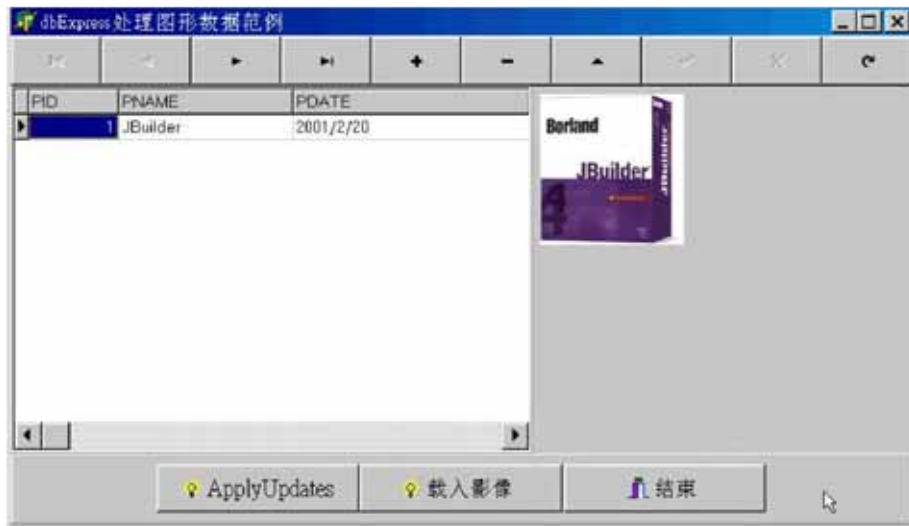


图8-7 范例应用程序加载JPEG类型数据的画面

8.3 如何有效率地处理二进制大型数据

除了处理前面讨论的一般图形数据之外，许多应用程序也需要处理大量的 **BLOB** 数据，例如动画类型的数据，**AVI**类型的动画文件便是体积庞大的 **BLOB**数据。当需要处理这种巨大的数据时，如果不注意应用程序的性能，那么当读者使用一般的 **dbExpress**技术开发完应用程序之后，可能会发现应用程序的性能令人不能接受。如果读者碰到这种情形，那么应该怎么办呢？

读者必须知道，当处理数量众多的巨量 **BLOB**数据时必须特别小心。例如 **AVI**文件的大小可能动辄数百 **KB**，甚至数 **MB**。因此如果读者一次处理数百笔这种数据时，就算不会发生内存不足的问题，也会让应用程序慢到似乎是死机的程度。在这种情形中，读者除了应该按照前面章节说明的把 **TSQLClientDataSet**的 **PacketRecords**特性值设置为一个正数值（例如10）以便一次只读取和处理几个记录之外，也必须采用优化处理 **BLOB**数据的方式来开发这种应用程序，这样应用程序的性能才不会损失太多。

本小节的目的在于说明如何有效率地处理 **BLOB**类型的数据，让读者了解在处理大量 **BLOB**数据时应该使用的技巧。本章将以开发一个处理 **AVI**动画文件的范例来说明如何处理数量众多而且庞大的 **BLOB**数据的应用系统。

请使用下面的步骤建立范例应用程序：

- 1) 在 **Delphi**集成开发环境中点击 **File New Application**菜单建立一个新的项目。
- 2) 在主窗体中放入图8.8所示的各个可视化组件以及数据感知组件。
- 3) 点击 **File New Data Module**以建立一个数据模块，并且在数据模块中放入

TSQLConnection、TSQLDataSet、TDataSetProvider和TClientDataSet组件，如图8 9所示。

4) 双击TSQLConnection组件以激活dbExpress连接对话框，并且将它连接到范例数据库D7Books.GDB。

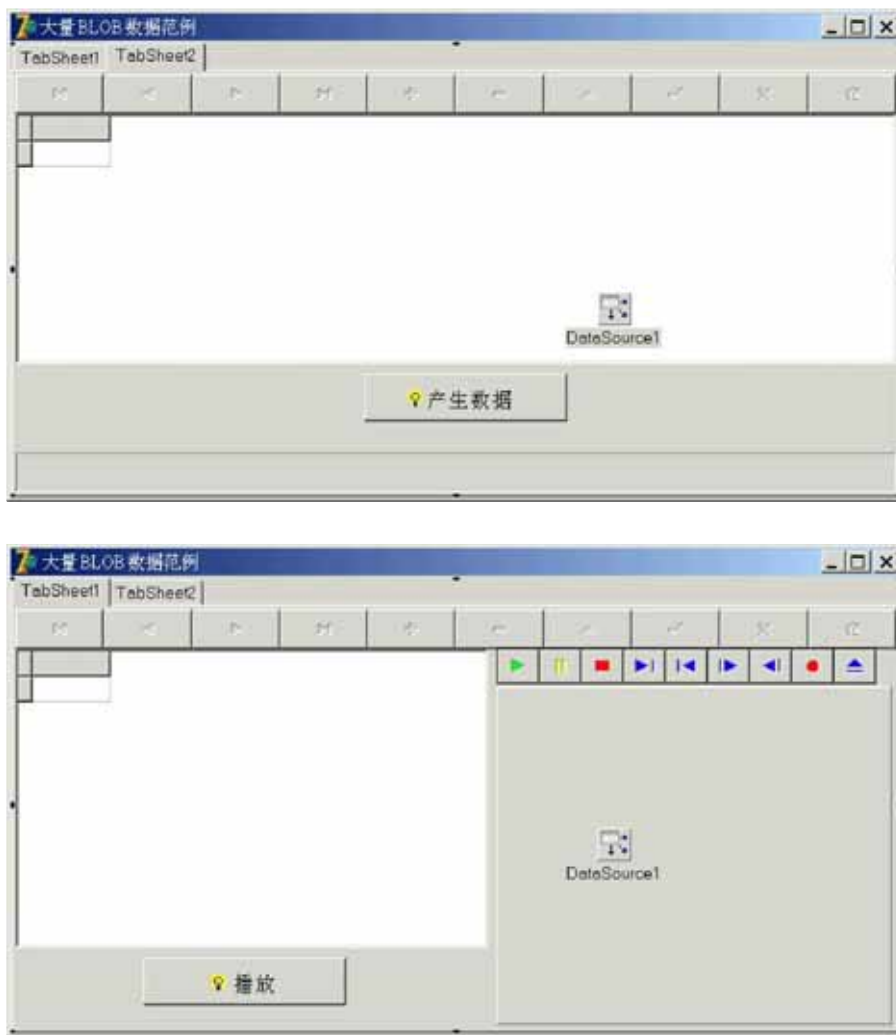


图8-8 范例应用程序的主窗体

图8 10是dbExpress连接对话框的画面。在这个对话框中有一个 BlobSize特性。这个特性值的大小与 dbExpress应用程序如何处理 BLOB类型的数据有关。BlobSize的默认值为 1，代表 dbExpress不限制每一个记录能够处理的 BLOB数据大小。BlobSize特性值可以设置为 1、0和一个正数，下列的表格总结了这些不同数值的意义。

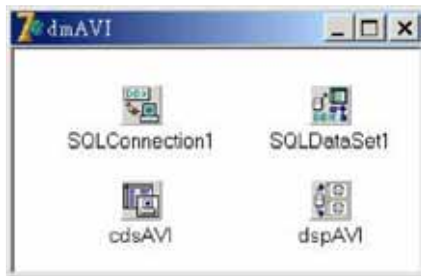


图8-9 范例应用程序的数据模块和 dbExpress组件

BlobSize特性值	意 义
1	代表对记录的 BLOB 数据不设大小限制，但是 dbExpress 会先预留一定的缓冲区来处理 BLOB 数据。如果 BLOB 数据小于这个预定的缓冲区，那么便会发生资源浪费的情形
0	代表 dbExpress 配置的缓冲区匹配每一个记录真正的 BLOB 数据大小，不会浪费资源，也不会发生数据被截短的情形
正数	代表每一个记录的 BLOB 数据大小就是这个正数的设置值，dbExpress 只会配置这么大的缓冲区来处理 BLOB 数据。如果记录的 BLOB 数据超过此大小，那么 BLOB 数据超过的部分便会被截短

在这个范例应用程序中，由于需要处理大量的 BLOB 数据，因此我们并不希望浪费任何资源，而希望 dbExpress 根据真正的 BLOB 数据大小来配置缓冲区，所以正确的设置值应该是 0。当然，如果应用程序处理的每一个 BLOB 数据的大小都是固定的，例如 32KB，那么读者可以设置 BlobSize 为 32 以便有效率地处理 BLOB 数据。

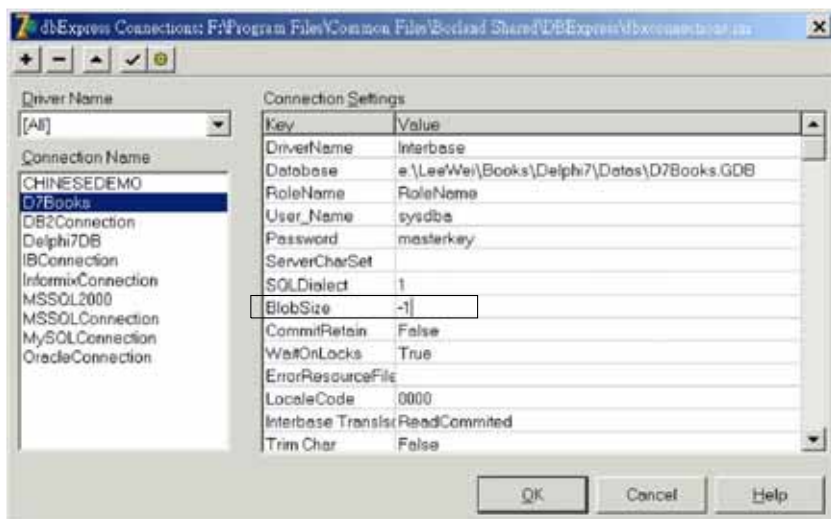


图8-10 设置 BlobSize 为 0

5) 设置 TSQLConnection 组件的 BlobSize 大小为 0，并且把 TSQLConnection 的

Active设置为True。

6) 设置TSQLDataSet的DBConnection特性值为刚才放入的 TSQLConnection，再设置TSQLDataSet的CommandText特性值为select * from LARGEAVI。

7) 设置TDataSetProvider的DataSet特性值为数据模块中的 TSQLDataSet。

8) 最后 设置 TClientDataSet的 ProviderName特性值为数据模块中的 TDataSetProvider组件。

当应用程序处理包含大量 BLOB数据的记录时，如果用户并没有使用每一个记录中的BLOB数据，那么事实上应用程序根本不需要从后端数据源中取得 BLOB数据并且传递到客户端，而是应该只在有用户真的访问 BLOB数据时才从后端数据源中取得特定的BLOB数据，这样可以大幅增加应用程序的性能。

TDataSetProvider组件的Options特性值中的poFetchBlobsOnDemand特性值便控制了这样的执行行为。它的默认值为 False，代表dbExpress会把所有BLOB数据都访问到客户端，而不管用户是否真的使用这些 BLOB数据。因此为了让范例应用程序能够更有效率地处理BLOB数据，让我们设置这个特性的数值为 True，以便让范例应用程序在真的需要BLOB数据时才从后端数据库中取得 BLOB数据。

9) 设置TDataSetProvider的poFetchBlobsOnDemand特性值为 True，以避免访问不使用的BLOB数据（见图8 11）。

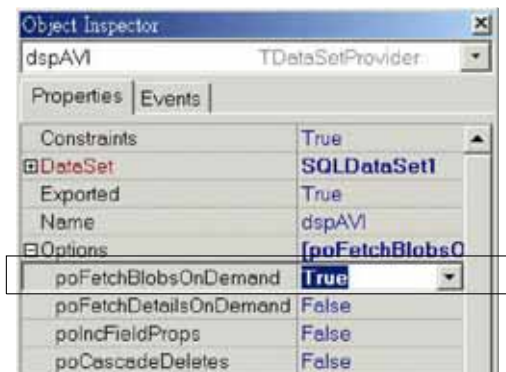


图8-11 设置TDataSetProvider的poFetchBlobsOnDemand以避免访问不使用的BLOB数据

本范例的其余步骤就是实现这个范例，并且观察经过上述的设置调整之后是否有比较好的性能。

10) 首先是实现图8 8中“产生数据”按钮的 OnClick事件处理函数。下面就是和它相关的实现程序代码：

```
const
    LOOP = 50;

procedure TForm1.bbtnGenerateClick(Sender: TObject;
var
```

```

    iCount : Integer;
begin
    pbCount.Max := LOOP;
    pbCount.Position := 0;

    lStart := GetTickCount;

    for iCount := 1 to LOOP do
    begin
        iAVI := Random(4);
        if (iAVI = 3) then
            iAVI := Random(4);

        dmAVI.cdsAVI.Insert;
        dmAVI.cdsAVI.FieldByName('ID').Value := GetID;
        dmAVI.cdsAVI.FieldByName('NAME').Value := GetName;
        dmAVI.cdsAVI.FieldByName('INDATE').Value := Now;
        GetAVI;
        dmAVI.cdsAVI.Post;

        pbCount.Position := pbCount.Position + 1;
        Application.ProcessMessages;
    end;
    dmAVI.cdsAVI.ApplyUpdates(0);
    lEnd := GetTickCount;

    Self.Caption := FloatToStr((lEnd - lStart) / 1000.0);
end;

procedure TForm1.GetAVI;
begin
    dmAVI.cdsAVI.LoadFromFile(dmAVI.cdsAVI.FieldByName('NAME').AsString);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Randomize;
    iIID := 1;
    aviFiles[0] := 'COOL.AVI';
    aviFiles[1] := 'Count24.AVI';
    aviFiles[2] := 'dx.AVI';
    aviFiles[3] := 'Speedis.AVI';
end;

```

上面的程序代码使用随机的方式把四个范例 AVI文件加载到 LARGEAVI数据表中。而这四个 AVI文件是大小不一的,从最小的 35KB到最大的4MB多,这主要是为了稍后观察范例应用程序的执行行为,以及考验 dbExpress处理大小不同的 BLOB数据的能力。

11) 接着实现图8 8中的“播放”按钮的 OnClick事件处理函数。下面就是它和相关的实现程序代码:

```
procedure TForm1.bbtnPlayClick (Sender: TObject);
begin
    GetStartTime;

    dmAVI.cdsAVIAVI.SaveToFile ('TEMPAVI.AVI');
    MediaPlayer1.FileName := 'TEMPAVI.AVI';
    MediaPlayer1.Open;
    MediaPlayer1.Play;

    GetEndTime;
    ShowRunTimeMsg;
end;

procedure TForm1.MediaPlayer1Notify (Sender: TObject);
begin
    if (MediaPlayer1.Mode = mpStop) then
    begin
        MediaPlayer1.Close;
    end;
end;

procedure TForm1.GetEndTime;
begin
    lEnd := GetTickCount;
end;

function TForm1.GetRunTime: Double;
begin
    Result := (lEnd - lStart) / 1000.0;
end;

procedure TForm1.GetStartTime;
begin
    lStart := GetTickCount;
end;
```

```
procedure TForm1.ShowRunTimeMsg;  
begin  
    Self.Caption := Self.Caption + '-' + FloatToStr(RunTime);  
end;
```

“播放”按钮的OnClick事件处理函数主要是把LARGEAVI数据表中存储的AVI字段内容存储到一个临时文件中，再使用TMediaPlayer来播放这个AVI文件，并且计算这个过程需要的时间。由于是从数据表中取得BLOB数据，因此可以测试dbExpress在处理BLOB数据时的效率。

现在就可以执行范例应用程序，看看它在处理大量的BLOB数据时通过设置TSQLConnection的BlobSize和TDataSetProvider的Options\poFetchBlobsOnDemand特性值是否会产生不同的效果。

图8 12和图8 13分别是设置poFetchBlobsOnDemand为True和False时，在“产生数据”按钮被点击随机产生数据之后的结果。从图和对比数据表格中可以看到，当开启拥有大量BLOB数据的数据表时，设置poFetchBlobsOnDemand为True可以大幅增加应用程序的反应速度和性能，并且能够有效地减少客户端应用程序对于内存资源的需求。在下面poFetchBlobsOnDemand设置为False的情形中，TClientDataSet会试着把所有BLOB数据一次加载到客户端，这样做不但缓慢和浪费资源，甚至会造成客户端死机的情形，因为所有客户端的资源可能被消耗光。

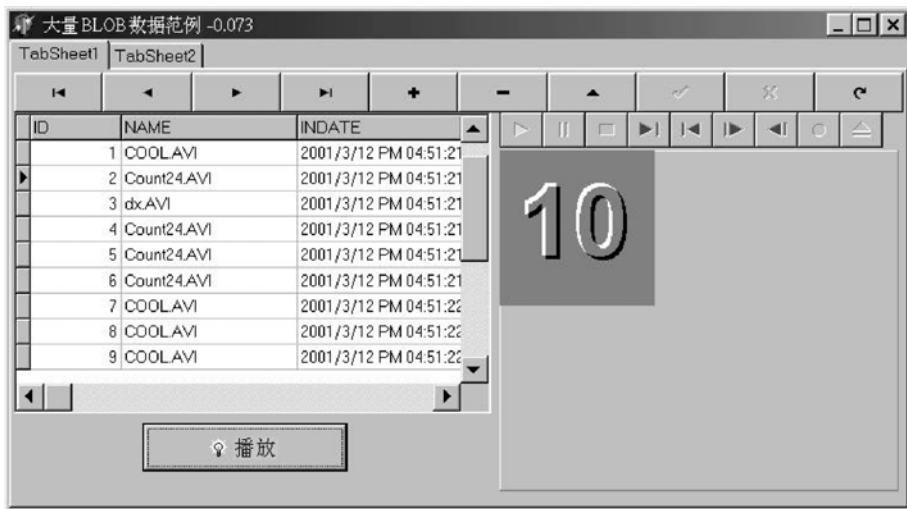


图8-12 设置TDataSetProvider的poFetchBlobsOnDemand为False情形

	需要BLOB数据时才访问	一开始就访问BLOB数据
时间	0.073	6.706

相反，设置poFetchBlobsOnDemand为True则会让TClientDataSet只取得其他字段

的数据，而不访问 BLOB 的数据。BLOB 的数据只有在客户端应用程序真正需要时才会由 dbExpress 自动加载到客户端。

当然，在设置 poFetchBlobsOnDemand 为 True 的情形中读者可能会质疑如此一来会不会降低客户端应用程序播放 AVI 文件的效率。这是一个好问题，现在就让我们直接使用范例应用程序来测试这种情形。

图8 14就是点击“播放”按钮播放数据表中存储的 AVI 文件的结果。

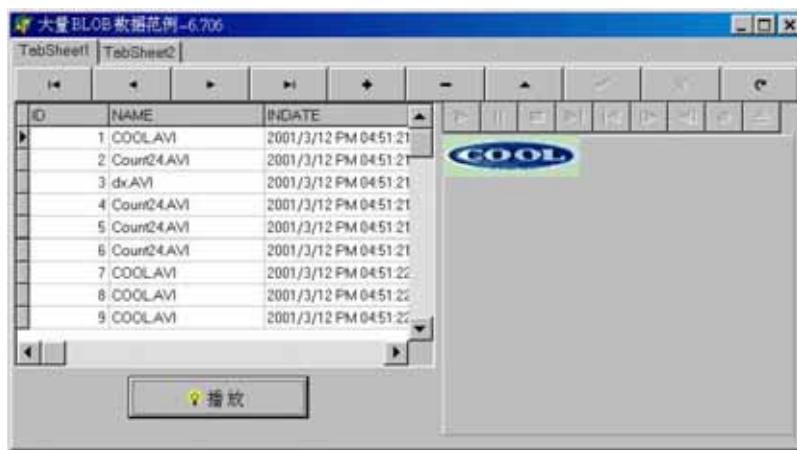


图8-13 设置TDataSetProvider的poFetchBlobsOnDemand为True情形

下面的表格总结了当范例应用程序播放 500KB 大小的 AVI 文件时，设置 poFetchBlobsOnDemand 为 True 和 False 的结果：

	需要BLOB数据时才访问	一开始就访问BLOB数据
播放BLOB数据时间（500KB）	0.519	0.552

从上面表格中的数据可以知道，在设置 poFetchBlobsOnDemand 为 True 的情形中，当 BLOB 数据的大小在 1.2MB 左右时，还是比设置 poFetchBlobsOnDemand 为 False 好。这实在很令人满意。这个现象一直要到 BLOB 数据超过 2MB 时才会改变。

例如，下面的图 8 15 和表格数据是范例应用程序播放 4MB 大小的 AVI 文件时的执行结果。

当 BLOB 数据的大小超过 2MB 时，由于在设置 poFetchBlobsOnDemand 为 False 的情形中这些数据已经被加载到客户端，因此它的性能比较好。

	需要BLOB数据时才访问	一开始就访问BLOB数据
播放BLOB数据时间（4M多）	1.965	0.653

不过即使如此，在设置 poFetchBlobsOnDemand 为 True 的情形中需要的时间也只是 2 秒左右，这还是可以让人接受的。

上面的执行结果是在单机中执行的结果，读者可以在网络中测试范例应用程序的执行状态。

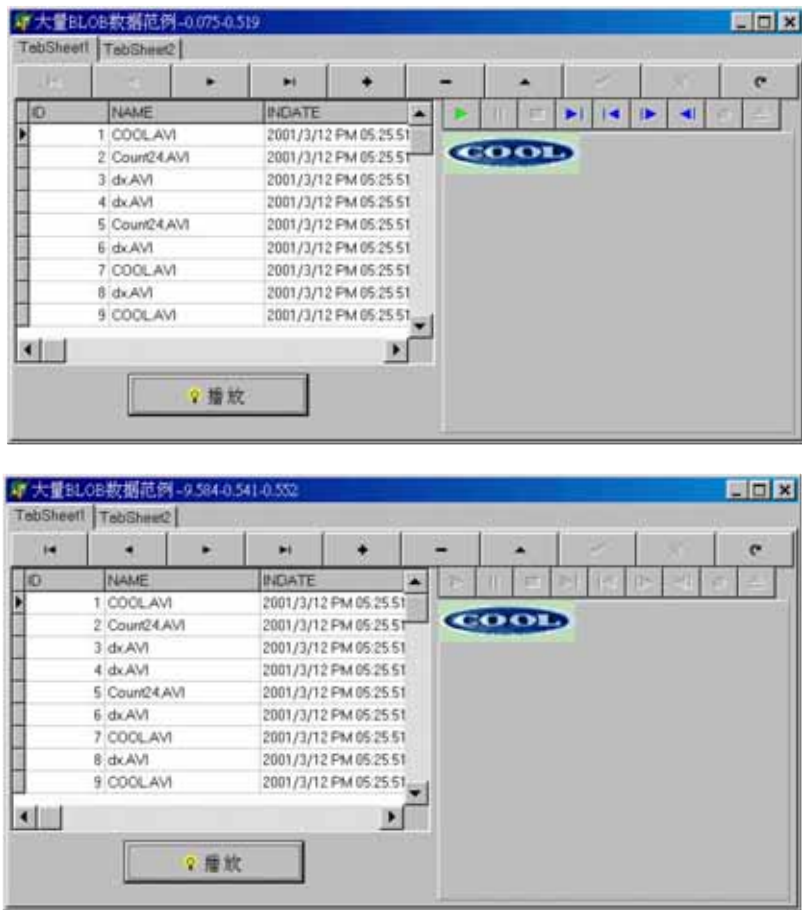


图8-14 播放不同大小的AVI文件的结果

从这个范例中，我们可以得到下面的结果：

- 设置TSQLConnection的BlobSize为0可以增加应用程序的效率，也可以减少浪费。
- 设置TDataSetProvider的Options\poFetchBlobsOnDemand可以加快应用程序的响应速度。
- 当BLOB数据的大小在 2MB以下时，设置 TDataSetProvider的Options\poFetchBlobsOnDemand特性值为True不但可以增加性能，也可以避免浪费客户端应用程序的资源。

一般来说，设置 TSQLConnection的BlobSize为0并且设置 TDataSetProvider的Options\poFetchBlobsOnDemand为True可以让应用程序在处理大量 BLOB数据时比较

有效率,也比较稳定。读者在使用 dbExpress开发应用程序时应该尽量利用 dbExpress的这个特性。

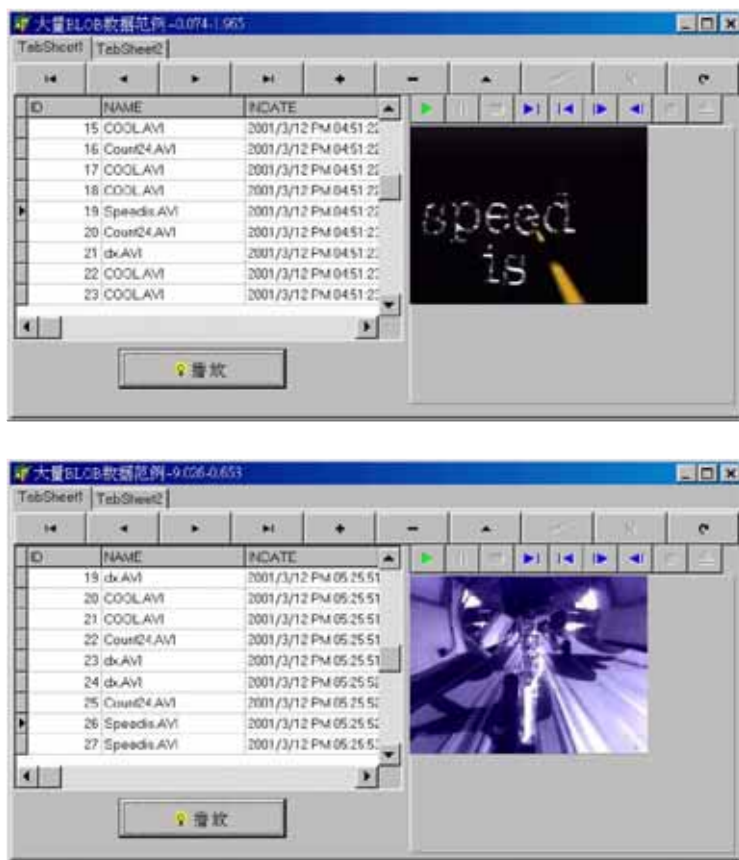


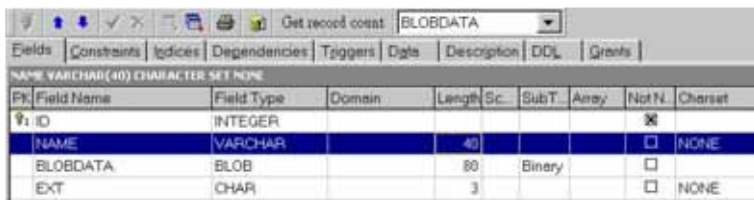
图8-15 播放不同大小的AVI文件的结果

8.4 OLE Container类型的数据

除了处理一般的图形数据之外, dbExpress也能够结合 Windows的OLE功能在数据库中存储各种不同的 BLOB数据,并且在显示这些 BLOB数据时自动激活可以处理特定BLOB数据的应用程序。例如如果数据表字段中存储的 BLOB数据是 Word文件,那么在用户双击此文件之后,应用程序便可以激活 Word来编辑这份文件。本小节讨论的内容就是描述如何结合 dbExpress和OLE的功能来开发处理OLE文件的应用程序。

由于本小节的范例使用了 Windows的OLE功能,因此这些应用程序只能在 Windows平台上执行,而无法在 Linux平台上执行,这是读者需要注意的。

在本书的范例数据库中包含了一个 **BLOBDATA** 数据表，这个数据表的纲要如图 8-16 所示。



Field Name	Field Type	Domain	Length	Sc.	SubT.	Array	Not N.	Charset
ID	INTEGER						<input checked="" type="checkbox"/>	
NAME	VARCHAR		40				<input type="checkbox"/>	NONE
BLOBDATA	BLOB		80		Binary		<input type="checkbox"/>	
EXT	CHAR		3				<input type="checkbox"/>	NONE

图8-16 范例数据表的字段结构

在 **BLOBDATA** 数据表中包含了一个 **BLOBDATA** 字段，这个字段的类型是 **BLOB**。在我们即将实现的范例中将在这个字段中存储各种不同的 **BLOB** 数据，例如 Word 文件、AVI 文件或是 PowerPoint 文件等。

步骤1：设计范例应用程序

首先在 Delphi 集成开发环境中建立一个新的应用程序项目，再建立一个数据模块，并且在此数据模块中放入 **TSQLConnection** 组件，连接到范例数据库 **D7Books**，再放入 **TSimpleDataSet**，最后放入一个 **TSQLDataSet** 组件，如图 8-17 所示。



图8-17 在数据模块中放入相关的dbExpress组件，以访问范例数据表

接着设置刚才放入的组件的特性值：

TDataModule:

特性名称	设置特性值
Name	dmOLEContainer

TSQLConnection:

特性名称	设置特性值
Name	SqlcOLE
ConnectionName	D7Books

TSQLClientDataSet:

特性名称	设置特性值
Name	scdsBLOBData
CommnadText	select * from BLOBDATA

TSQLDataSet:

特性名称	设置特性值
Name	sdsMaxID
ConnectionName	select MAX (ID) from BLOBDATA

scdsBLOBData的目的是从BLOBDATA数据表中取得所有数据，而 sdsMaxID则是取得目前在BLOBDATA数据表中ID字段的最大值。

接着在范例应用程序的主窗体中放入图 8 18所示的控件，包括 TDataSource、TDBNavigator、TDBGrid、TOpenDialog、三个TBitBtn组件以及一个TOleContainer组件。之后，请设置这些组件的特性值：

TDataSource:

特性名称	设置特性值
Name	dsBLOBDATA
DataSet	dmOLEContainer.scdsBLOBData

TDBNavigator:

特性名称	设置特性值
Name	DBNavigator1
DataSource	dsBLOBDATA

TDBGrid:

特性名称	设置特性值
Name	DBGrid1
DataSource	dsBLOBDATA

TOpenDialog:

特性名称	设置特性值
Name	OpenDialog1

TBitBtn:

特性名称	设置特性值
Name	bbtnLoad
Caption	加载对象

TBitBtn:

特性名称	设置特性值
Name	bbtnShow
Caption	显示BLOB内容

TBitBtn:

特性名称	设置特性值
Name	bbtnApplyUpdate
Caption	ApplyUpdates

TOleContainer:

特性名称	设置特性值
Name	OleContainer1

在设置了主窗体中放入的组件之后，下一步就是开始实现此范例应用程序。

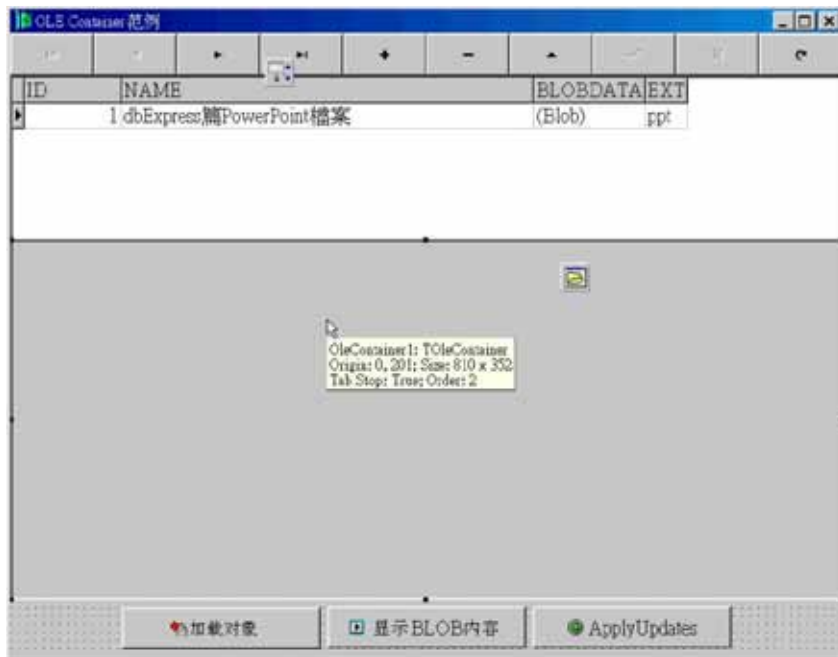


图8-18 范例应用程序的主窗体，其中使用了TOleContainer组件

步骤2：实现范例应用程序的功能

在主窗体中“加载对象”按钮的功能是，让用户在新增或修改数据时能够点击此按钮来加载BLOBDATA字段存储的BLOB数据。请双击此按钮并且在它的OnClick事

件处理函数中编写如下的程序代码：

```
procedure TForm1.bbtnLoadClick(Sender: TObject);
var
    fnOle : String;

function BlobContentToString(const sFileName : String): String;
begin
    with TFileStream.Create(sFileName, fmOpenRead) do
        begin
            try
                SetLength(Result, Size);
                Read(Pointer(Result)^, Size);
            finally
                Free;
            end;
        end;
    end;

begin
    if (OpenDialog1.Execute) then
        begin
            fnOle := OpenDialog1.FileName;
            dmOLEContainer.scdsBLOBData.Edit;
            dmOLEContainer.scdsBLOBData.FieldName('BLOBDATA').AsString :=
                BlobContentToString(fnOle);
            dmOLEContainer.scdsBLOBData.FieldName('EXT').Value :=
                GetFileExtension(fnOle);

            dmOLEContainer.scdsBLOBData.Post;
        end;
    end;
```

bbtnLoadClick事件处理函数首先调用 OpenDialog1的Execute方法以便开启文件对话框让用户选择要加载进 BLOBDATA字段的数据。在用户成功地选择了欲加载的文件之后，bbtnLoadClick先进入修改模式，接着调用 BlobContentToString函数把被加载的文件内容转换为字符串流的形式，再把转换后的结果存储在 BLOBDATA字段中，并且在EXT字段中存储加载文件的扩展名，最后调用 scdsBLOBData的Post方法把修改的结果写回数据表中。而 BlobContentToString函数使用 TFileStream加载用户选择的文件，并且存储在返回的字符串流中。

在实现了将 BLOB数据存储到 BLOB字段中之后，再看看如何实现显示 BLOB字

段内容的功能。当用户点击了主窗体中的“显示 BLOB 内容”按钮之后，主窗体中的 TOleContainer 组件便会显示 BLOBDATA 字段的内容。BbtnShow 的 OnClick 事件处理函数的实现如下：

```

procedure TForm1.bbtnShowClick (Sender: TObjedt;
var
    sFileName : String;
    bs : TClientBlobStream;
begin
    bs := TClientBlobStream.Create(dmOLEContainer.scdsBLOBDataBLOBDATA, bInRead
    try
        sFileName := ExtractFilePath(Application.ExeName) + 'tempBlob';
        sFileName := sFileName + '.' +
dmOLEContainer.scdsBLOBData.FieldName('EXT').AsString;
        bs.Seek (soFromBeginning, 0);
        bs.SaveToFile (sFileName);
        OleContainer1.CreateObjectFromFile (sFileName, False
    finally
        bs.Free;
    end;
end;

function TForm1.GetFileExtension (const sFileName: String String;
begin
    Result := Copy(sFileName, Length(sFileName) - 2, 3);
end;

procedure TForm1.BitBtn1Click (Sender: TObjedt;
begin
    dmOLEContainer.scdsBLOBData.ApplyUpdates (0);
end;

```

bbtnShowClick 首先建立 TClientBlobStream 组件，并且把 BLOBDATA 数据表目前记录的 BLOBDATA 字段的内容传入。接着 bbtnShowClick 会在目前的执行目录下把 TClientBlobStream 组件的内容存储在一个暂时的文件中，最后 bbtnShowClick 调用 OleContainer1 的 CreateObjectFromFile 方法把暂时的文件内容显示在主窗体中的 OleContainer1 组件上。

下面的实现程序代码当用户在主窗体的 DBGrid1 中新增数据时自动设置 ID 字段的数值。当然，如果读者使用的数据库拥有 AutoInc 类型的字段，那么就无需使用下面的程序代码。

```

function TdmOLEContainer.GetMaxID: Integer;
begin

```



```
try
    sdsMaxID.Open;
    try
        Result := sdsMaxID.Fields[0].Value + 1;
    except
        on Exception do
            Result := 1;
    end;
finally
    sdsMaxID.Close;
end;
end;
end;

procedure TdmOLEContainer.scdsBLOBDataBeforePost (DataSet: TDataSet;
begin
    if (VarIsNull (DataSet.FieldByName ('ID') .Value)) then
        DataSet.FieldByName ('ID') .Value := GetMaxID;
end;
```

现在这个范例应用程序已经完成了，接着就让我们执行它并且开始加入数据，看看这个范例应用程序的执行结果。

步骤3: 执行范例应用程序并且观察执行结果

请编译并且执行范例应用程序，如图 8 19所示在DBGrid1中加入两个记录，并且分别在第一个和第二个记录的 BLOBDATA字段中点击“加载对象”按钮来加载一个PowerPoint文件和Word文件。而当我们浏览到包含 Word文件的记录并且点击了“显示BLOB内容”按钮后，主窗体下方的 OleContainer1便会显示BLOBDATA字段中的内容。

如果此时用户使用鼠标双击 OleContainer1，那么 OleContainer1便会根据目前 BLOBDATA字段的数据类型自动激活相关的应用程序来处理，例如图 8 20显示了当笔者双击了OleContainer1之后，范例应用程序便会自动激活 Word来处理这个Word文件，而在OleContainer1中内嵌了Word应用程序。

现在让我们在BLOBDATA中加入第3个记录，并且在BLOBDATA字段中加载AVI的文件，那么点击“显示 BLOB内容”按钮后AVI的第一个画面便会显示在 OleContainer1中。如果继续双击OleContainer1，范例应用程序便会激活 MediaPlayer开始播放AVI文件的内容，如图8 21所示。

这个范例展示了如何结合 dbExpress和OLE的功能，读者可以在自己开发的应用程序中使用类似的技巧。事实上这个范例还不算完全实现了，例如在图 8 20中如果用户激活 Word应用程序并且编辑了 BLOBDATA字段中的Word文件，那么这份最新的Word文件应该更新回数据表中。这个工作很简单，因此就留给读者来完成吧。



图8-19 范例应用程序的执行画面



图8-20 双击图8-19下方的Word文件，TOleContainer便会激活Word让用户进行编辑



图8-21 当BLOB内容是AVI文件时，TOleContainer便会使用MediaPlayer播放AVI内容

8.5 结论

由于BLOB类型的数据可以代表图形、动画、声音以及巨量的文本等信息，因此

许多数据库应用程序都需要处理这种类型的数据。本章说明了如何使用 **dbExpress** 来处理 **BLOB** 类型的数据，并且以范例的方式说明了如何使用 **dbExpress** 来开发处理 **BLOB** 数据的应用程序，让读者能够快速地学习到如何应用 **dbExpress** 处理图形和大量 **BLOB** 数据的技术。

除此之外，**Microsoft** 在 **Windows** 平台中提供了 **OLE** 的功能，能够让应用程序自动地处理特定类型的数据。**dbExpress** 虽然是平台中立的数据访问技术，但是 **dbExpress** 仍然能够轻易地结合平台特定的功能来进行更好的运用。因此本章也讨论了如何在 **Windows** 平台中结合 **dbExpress** 以及 **OLE** 的功能来开发应用程序。

此外，**BLOB** 类型的数据一般来说都比较大，因此当处理 **BLOB** 类型的数据时会对应用程序的性能造成影响，特别是当需要处理数量众多的记录时。本章的重点除了说明如何使用 **dbExpress** 处理 **BLOB** 数据之外，更重要的是也详细讨论了如何有技巧地使用 **dbExpress** 来有效率地处理 **BLOB** 类型的数据。通过了解 **dbExpress** 中与 **BLOB** 数据有关的设置以及运用原理之后，读者便可以很方便而且正确地使用 **dbExpress** 开发 **BLOB** 类型的应用程序了。

第四部分

深入的dbExpress实战技术



第9章 dbExpress和元数据

在前面的章节中，本书讨论了如何使用 **dbExpress** 来处理“一般的数据”。所谓的“一般的数据”是指用户或是应用系统使用的数据，程序员使用 **dbExpress** 提供的功能来处理这种数据以满足用户的需求。不过除了用户和应用系统使用的数据之外，事实上在数据库中还有另外一种“系统数据”。所谓的“系统数据”就是描述数据库本身中存有什么数据，以及每一种数据的详细信息等。例如，这些“系统数据”描述了目前在数据库中有哪些数据表，每一个数据表拥有的字段名称、字段类型以及数据表定义的索引信息等。这些属于数据库的“系统数据”便统称为“元数据”。

本章要讨论的内容就是说明如何使用 **dbExpress** 来处理元数据。也许读者会觉得元数据应该是属于 **dbExpress** 底层的信息和技术，一般的数据库应用程序应该不需要处理元数据。的确，元数据在大多数的应用中很少被使用，而且即使需要访问元数据也可以直接使用 **dbExpress** 组件来代劳，似乎不需要使用什么特别的技术。不过在读者读完本章之后却会发现元数据不仅可以在应用程序中使用，对于元数据有正确的认识也可以帮助读者掌握 **dbExpress** 应用程序的性能。

9.1 dbExpress和元数据

dbExpress 在内部许多地方都使用了元数据，元数据用来处理各种数据的应用，包括让客户端的 **TClientDataSet**/**TSimpleDataSet** 根据元数据产生正确的 SQL 语句以选取和修改数据、访问数据表的索引信息等。**dbExpress** 使用这些元数据完成用户要求的工作。

同样，程序员也可以使用元数据来处理应用上的需要，例如 **Delphi** 的 **SQL Explorer** 便使用了元数据让用户操作数据库中的各种对象。例如，在使用 **SQL Explorer** 开启 **MS SQL Server** 时，**SQL Explorer** 便使用元数据显示数据库中的数据表、存储过程、索引等可供操作和使用的对象（见图 9.1）。

许多应用程序也使用元数据来开发各种应用，例如程序员可以使用元数据编写类似 **SQL Explorer** 的工具以便管理数据库应用系统，或是使用元数据访问索引信息允许用户在应用程序执行时动态改变用于过滤数据的索引。

那么如何在 **dbExpress** 中访问和使用元数据呢？这并不困难，因为 **dbExpress** 提供了数种方法允许程序员访问和处理元数据。最简单的方式是直接使用 **dbExpress** 组件提供的方法。例如，**TSQLConnection** 组件便提供了许多方法让程序员访问元数据，

在下一小节的范例中读者便可以看到如何使用 **dbExpress** 组件访问元数据。另外一种方式是直接使用 **dbExpress** 的标准接口来访问元数据，这种方式拥有最大的弹性和功能，但是要求程序员对于 **dbExpress** 标准接口有一定的了解，并且能够熟练地使用接口程序设计技巧。

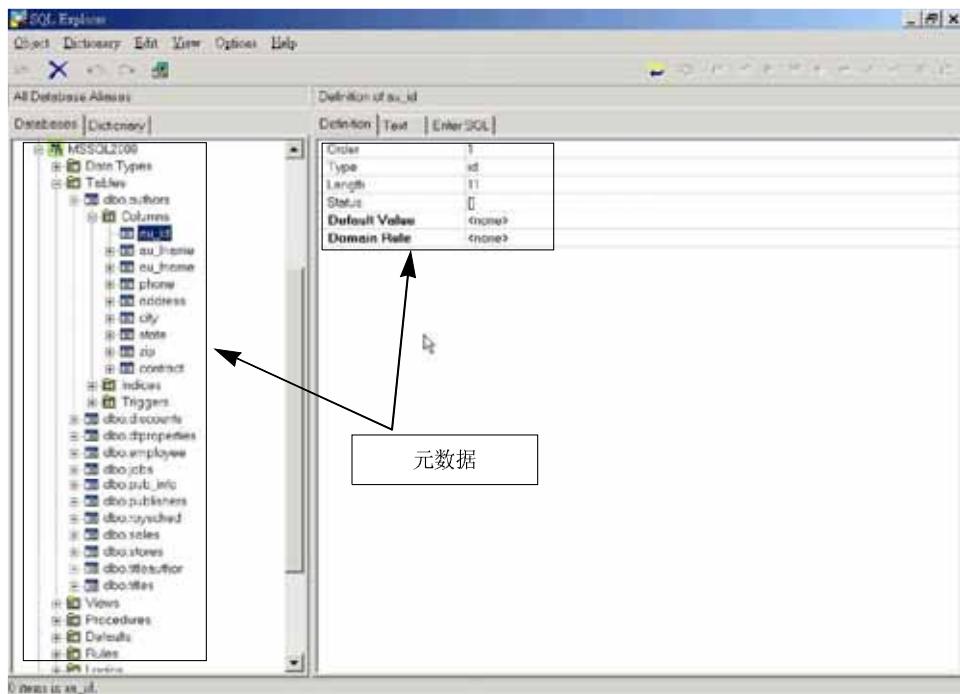


图9-1 SQL Explorer使用元数据呈现数据库中的信息

有关 **dbExpress** 标准接口的内容读者可以在第 13 章中看到详细的讨论，在本章中我们将采用 **dbExpress** 组件以及自行开发的 **Object Pascal** 类来访问和处理元数据。

9.2 使用dbExpress处理元数据

本小节将讨论如何使用 **dbExpress** 组件来处理元数据，由于 **dbExpress** 组件已经提供了一些方法，程序员可以通过调用它们使用元数据，因此本小节将直接使用这些方法。不过由于 **dbExpress** 组件提供的方法在使用上仍然比较复杂，因此本小节将编写一个简单的 **Object Pascal** 类来帮助我们访问和处理元数据。读者将会看到这个类可以大大简化客户端的程序代码。当然，读者在了解了这些程序代码的意义之后也可以再在这个类中加入更多的功能。

在讨论如何处理元数据之前，我们仍然必须了解在 **dbExpress** 中已经提供了哪些可以帮助程序员处理元数据的定义以及元数据的分类，这样我们就可以直接使用由

dbExpress提供的基础功能，而无需重复开发。

1. dbExpress定义的元数据种类

在dbExpress中定义了数个常数来代表不同的元数据，当程序员欲使用 dbExpress 组件访问元数据时，必须传入适当的常数定义以便告诉 dbExpress想要访问的元数据种类。下面的表格列出了这些定义的常数以及每一个常数的意义：

常数数值	意 义
stNoSchema	不返回元数据。SQL数据集会包含由SQL语句或是存储过程产生的结果数据集（一般正常的数据库）
stTables	符合SQL Connection的TableScope特性指定的所有属于特定数据库的数据表信息
stSysTables	有关数据库使用的系统数据表的信息。如果数据库没有使用系统数据表存储信息，那么便会返回空白的数据集
stProcedures	有关数据库中所有存储过程的信息
stColumns	有关一个数据表的所有字段的信息
stProcedureParams	一个存储过程的所有参数的信息
stIndexes	一个特定数据表的所有索引的信息

这些常数的名称都以它代表的元数据种类来定义，例如从 stIndexes这个名称可以推知这个常数用来访问数据表的索引信息的元数据。

在上面的表格中， stTables以及stSysTables常数可以控制dbExpress访问的数据表种类。由于数据库中存在什么数据表， dbExpress要访问哪一类的数据表也都属于元数据，因此客户端和程序员可以通过这两个常数来指定欲访问的数据表种类。例如， TSQLConnection组件的TableScope特性值便允许程序员指定客户端欲访问的数据表种类，程序员可以通过这个特性值来访问一般数据表、系统数据表或是别名数据表等各种不同的数据表（见图9 2）。

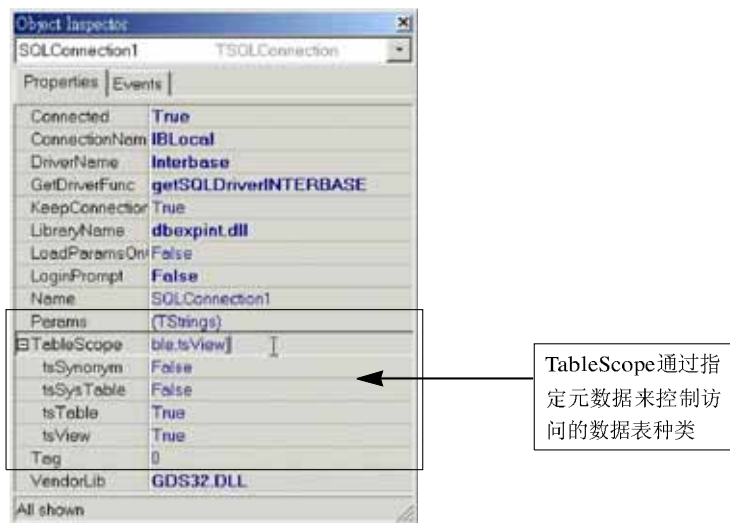


图9-2 TSQLConnection组件的TableScope特性值可以控制欲访问的数据表种类

当程序员使用上面表格中的常数访问元数据时，dbExpress在执行完毕之后会以结果数据集的形式返回这些元数据，程序员接着就可以按照处理一般数据的方式从返回的结果数据集中取得需要的元数据。例如，当客户端用 `stTables` 常数访问数据库中的所有数据表名称时，dbExpress便会把数据库中所有一般的数据表名称以结果数据集的形式返回给客户端，客户端只需要使用 `TClientDataSet/TSimpleDataSet` 组件接受此结果数据集，再通过 `Fields` 特性和 `Next` 等方法就可以取得数据库中的数据表名称了。

2. 处理元数据的范例 Object Pascal 类

在本章展示如何使用 dbExpress 以及 Object Pascal 程序代码处理元数据之前，让我们先开发一个 Object Pascal 类来处理许多一般而且例行的工作。如此一来就不需要在稍后的范例中一直重复这些程序代码。

本小节要开发的 Object Pascal 类允许客户端使用它来调用并且访问元数据。这个类根据上一小节列出的元数据常数定义了数个方法让客户端能够访问各种不同的元数据。客户端不需要知道如何使用这些常数以及如何调用 dbExpress 来取得元数据，只需建立这个类的对象并且调用其中的方法，就可以取得代表元数据的结果数据集。

首先，让我们为这个类取名为 `TdbExpressMetaData`，将它声明为从 `TComponent` 类继承下来的。由于 `TdbExpressMetaData` 在处理元数据时需要数个组件的帮助，因此我们在 `TdbExpressMetaData` 中声明了一个 `TClientDataSet` 对象变量、一个 `TSQLConnection` 对象变量以及一个代表要访问的数据表种类的变量。`TdbExpressMetaData` 类此时的程序代码如下所示：

```
unit uMetaData;

interface

uses
    Classes, Provider, DBClient, SqlExpr, Forms;

type
    TdbExpressMetaData class (TComponent)
    private
        FMetaDataSet: TClientDataSet;
        FSQLConnection: TSQLConnection;
        oldScope : TTableScopes;
```

接着我们就可以开始为 `TdbExpressMetaData` 类定义可供客户端调用的服务了。由于 `TdbExpressMetaData` 的服务是以上一小节中所列出的元数据控制常数为基础的，因此在下面的程序代码中 `TdbExpressMetaData` 的 `public` 声明部分定义了访问各种元数据的方法。另外，在 `TdbExpressMetaData` 的 `protected` 声明部分定义了一个公用的方法 `GetSchemaInfo`。`GetSchemaInfo` 是真正使用 dbExpress 访问和处理元数据的方法，

它会被public声明部分中的方法调用，这样public声明部分中的方法即可节省许多程序代码。

```
protected
    procedure GetSchemaInfo (SchemaType: TSchemaType; const SchemaObjectName,
        SchemaPattern: String);
public
    procedure GetTables (IncludeSysTables: Boolean = False);
    procedure GetViews;
    procedure GetSynonyms;
    procedure GetProcedures;
    procedure GetIndexes (const TableName: String);
    procedure GetColumns (const TableName: String);
    procedure GetNoSchema;
    procedure GetSystemTables;
    procedure GetProcedureParameters (const sProcedureName: String; published
    property SQLConnection: TSQLConnection read FSQLConnection write FSQLConnection;
    property MetaDataSet: TClientDataSet read FMetaDataSet write FMetaDataSet;
end;
```

在完成了TdbExpressMetaData声明的部分之后就可以进入实现的程序代码了。首先让我们看看 TdbExpressMetaData类最重要的公用方法 GetSchemaInfo。GetSchemaInfo使用TCustomSQLDataSet的SetSchemaInfo方法从后端数据源中访问元数据。TCustomSQLDataSet的SetSchemaInfo方法能够以结果数据集的形式返回访问的元数据，而且SetSchemaInfo方法在访问元数据时客户端能够通过 SetSchemaInfo的参数来指定欲访问的元数据种类。SetSchemaInfo的函数原型如下：

```
procedure SetSchemaInfo ( SchemaType: TSchemaType; SchemaObjectName,
    SchemaPattern: string);
```

SetSchemaInfo接受三个参数，第一个参数 SchemaType是上一小节列出的元数据控制常数，以指明客户端欲访问的元数据种类。第二个参数 SchemaObjectName控制欲访问的目标名称。例如，如果第一个参数 SchemaType是stColumns或是stIndexes，就代表要访问数据表的字段和索引信息，那么 SchemaObjectName就是欲访问的数据表名称。如果SchemaType被指定为stProcedureParams，那么SchemaObjectName就是欲访问的存储过程的名称。如果 SchemaType被指定为 stNoSchema、stTables、stSysTables或stProcedures，那么SchemaObjectName就无需指定数值。

第3个参数SchemaPattern是SQL的模式掩码（Pattern Mask），它可以控制返回的结果数据集如何过滤数据。任何不符合 SchemaPattern指定的模式的数据将不会出现在返回的结果数据集中。例如，假设现在我们要取得在数据库中所有以 P开头的数据库表名称，那么我们可以使用如下的程序代码：

```
SetSchemaInfo (stTables, '', 'P%');
```

其中SchemaPattern参数使用的‘%’字符是通配符，代表任何长度、任何字符的字符串。程序员也可以使用‘ ’字符来代表任何单一字符的信息。例如，如果程序员想要访问所有以PR开头而且以DUCT结尾的数据表名称，那么就可以使用如下的程序代码：

```
SetSchemaInfo (stTables, '', 'PR_DUQT');
```

如果程序员欲访问的元数据中存在‘%’或是‘ ’字符，那么程序员可以使用像C/C++一样的技术，以两个相同的‘%’或是‘ ’字符来代表单一的‘%’或是‘ ’字符。例如，程序员如果想要访问所有以PR开头而且以DUC%结尾的数据表名称，那么就可以使用如下的程序代码：

```
SetSchemaInfo (stTables, '', 'PR_DUC%%');
```

一般来说，在大多数的应用中，我们通常使用如下的程序代码，并不指明过滤条件以取得所有元数据信息。

```
SetSchemaInfo (SchemaType, SchemaObjectName);''
```

了解了如何使用SetSchemaInfo方法之后，下面是TdbExpressMetaData类的GetSchemaInfo方法的实现：

```
implementation

{ TdbExpressMetaData }

procedure TdbExpressMetaData.GetSchemaInfo (SchemaType: TSchemaType;
const SchemaObjectName, SchemaPattern: String;
var
cdsDataSet: TCustomSQLDataSet;
dspProvider: TProvider;
begin
cdsDataSet := TCustomSQLDataSet.Create(Self);
dspProvider := TProvider.Create(Self);
try
cdsDataSet.SQLConnection := SQLConnection;
cdsDataSet.SetSchemaInfo (SchemaType, SchemaObjectName, SchemaPattern);
dspProvider.DataSet := cdsDataSet;
MetaDataSet.Data := dspProvider.Data;
finally
dspProvider.Free;
cdsDataSet.Free;
end;
end;
```

在GetSchemaInfo中，首先建立暂时的TCustomSQLDataSet和TProvider组件，指定暂时TCustomSQLDataSet组件的TSQLConnection为目前应用程序使用的TSQL

Connection，接着调用SetSchemaInfo方法访问元数据。SetSchemaInfo使用的参数是GetSchemaInfo方法的参数，稍后我们会看到其他方法调用GetSchemaInfo方法时传递的参数。最后GetSchemaInfo方法把调用SetSchemaInfo之后取得的结果数据集通过TProvider组件存储在TdbExpressMetaData类公用的MetaDataSet对象变量中准备让客户端应用程序访问最终的元数据。

完成了GetSchemaInfo之后，我们就可以实现TdbExpressMetaData类的其他方法了，因为TdbExpressMetaData类的其他服务方法大都是使用GetSchemaInfo来取得元数据的。

首先是GetTables方法。当GetTables被调用之后，它先存储TSQLConnection原本的TableScope特性值，再根据输入参数IncludeSysTables的值来决定是否要取得系统数据表的元数据，再将tsTable以及tsSysTable值设置给TSQLConnection的TableScope特性值，最后调用GetSchemaInfo方法，同时传递三个参数（stables，‘，’）给GetSchemaInfo，代表要访问数据表信息。根据刚才对SetSchemaInfo的说明，由于要访问数据表的元数据，因此第2个参数无需设置数值，而且我们不使用任何过滤条件。在GetTables调用完GetSchemaInfo方法之后，包含数据表元数据的结果数据集就存储在TdbExpressMetaData对象的MetaDataSet对象变量中等待客户端访问了。

```
procedure TdbExpressMetaData.GetTables (IncludeSysTables: Boolean)
begin
    oldScope := SQLConnection.TableScope;
    if IncludeSysTables then
        SQLConnection.TableScope := [tsTable, tsSysTable]
    else
        SQLConnection.TableScope := [tsTable];
    GetSchemaInfo (stables, '', '');
    SQLConnection.TableScope := oldScope;
end;
```

GetViews方法使用的技巧和GetTables方法一样，只是以tsView作为传递给TSQLConnection的TableScope特性值。GetViews执行完毕之后，元数据也存储在TdbExpressMetaData对象的MetaDataSet对象变量中。

```
procedure TdbExpressMetaData.GetViews;
begin
    oldScope := SQLConnection.TableScope;
    SQLConnection.TableScope := [tsView];
    GetSchemaInfo (stables, '', '');
    SQLConnection.TableScope := oldScope;
end;
```

GetSynonyms方法将tsSynonym传递给TSQLConnection的TableScope特性值以访问别名数据表的信息，执行完毕之后元数据也存储在TdbExpressMetaData对象的

MetaDataSet对象变量中。

```
procedure TdbExpressMetaData.GetSynonyms;
begin
    oldScope := SQLConnection.TableScope;
    SQLConnection.TableScope := [tsSynonym];
    GetSchemaInfo (stTables, '', '');
    SQLConnection.TableScope := oldScope;
end;
```

GetProcedures方法更为简单，因为 GetProcedures访问数据库中所有存储过程的信息，因此它直接调用 GetSchemaInfo，并且传入stProcedures告诉SetSchemaInfo方法客户端欲访问存储过程信息。

```
procedure TdbExpressMetaData.GetProcedures;
begin
    GetSchemaInfo (stProcedures, '', '');
end;
```

GetIndexes方法访问特定数据表的索引信息，因此它在调用 GetSchemaInfo方法时除了传递stIndexes以指明要访问索引元数据之外，还需要通过第二个参数传递欲访问索引信息的数据表名称。

```
procedure TdbExpressMetaData.GetIndexes (const TableName: String;
begin
    GetSchemaInfo (stIndexes, TableName, '');
end;
```

GetColumns方法访问特定数据表的字段信息，例如字段的名称、字段的类型和字段的个数等。因此在调用 GetSchemaInfo时以stColumns作为第一个参数值，欲访问的数据表名称作为第二个参数值。

```
procedure TdbExpressMetaData.GetColumns (const TableName: String;
begin
    GetSchemaInfo (stColumns, TableName, '');
end;
```

GetSystemTables方法用来访问数据库中描述系统信息的系统数据表。GetSystemTables方法和GetTables不一样的地方是GetSystemTables只能访问系统数据表。它调用GetSchemaInfo并且传入stSysTables作为第一个参数值。

```
procedure TdbExpressMetaData.GetSystemTables;
begin
    GetSchemaInfo (stSysTables, '', '');
end;
```

最后的GetProcedureParameters方法用来访问特定存储过程的参数信息，客户端在使用GetProcedures方法取得了存储过程的元数据之后，便可再调用 GetProcedure

Parameters来取得存储过程完整的参数信息。GetProcedureParameters也调用GetSchemaInfo并且传入stProcedureParams作为第一个参数值以指定要访问存储过程参数信息的元数据，并且以欲访问的存储过程名称作为第二个参数值。

```
procedure TdbExpressMetaData.GetProcedureParameters (  
    const sProcedureName: String  
begin  
    GetSchemaInfo (stProcedureParams, sProcedureName);  
end;
```

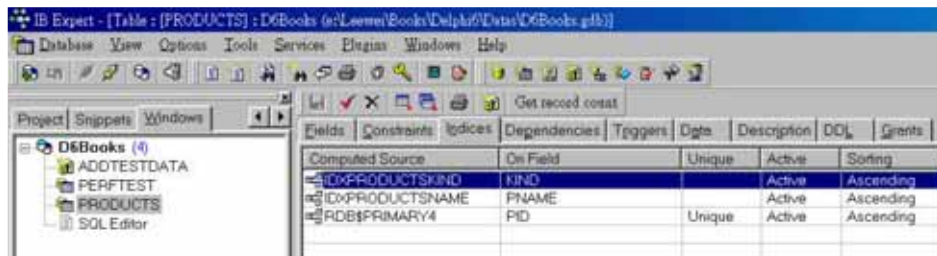
读者从上面的实现程序代码中可以发现，TdbExpressMetaData类的实现并不困难，但是有了TdbExpressMetaData类之后客户端访问元数据就非常容易了，只需要调用TdbExpressMetaData的服务方法，再访问TdbExpressMetaData的MetaDataSet特性即可顺利地取得元数据，因此接下来就让我们使用TdbExpressMetaData类来展示如何访问和处理数据库中的元数据。

3. 处理元数据的范例程序

现在我们就可以通过上一小节完成的TdbExpressMetaData类来说明如何使用dbExpress处理元数据了。读者会发现，在这些基本工作完成之后，处理元数据就像访问一般的数据一样简单。

本小节范例处理的目标是一个InterBase数据库，在这个InterBase数据库中定义了数个数据表，每一个数据表拥有许多字段，同时数据表也定义了索引，此外在这个InterBase数据库中也定义了存储过程。因此这个范例要展示的就是如何访问这些描述数据库对象的信息。一旦读者理解了这个范例，就可以继续进行扩充，将它开发成类似SQL Explorer的小工具并且附在应用系统中供用户使用。

图9 3和图9 4显示了范例InterBase数据库中的数据表信息，以及其中PRODUCTS数据表定义的字段和索引信息。



Fields	Constraints	Indices	Dependencies	Triggers	Data	Description	DDL	Grants
Computed Source	On Field	Unique	Active	Sorting				
ID-PRODUCTSKIND	KIND		Active	Ascending				
ID-PPRODUCTSNAME	PNAME		Active	Ascending				
IDB\$PRIMARY4	PID	Unique	Active	Ascending				

图9-3 范例InterBase中的数据表信息

现在先让我们看看如何访问这些字段和索引的元数据信息。首先在Delphi中建立一个应用程序，然后在主窗体中放入TSQLConnection、TSQLDataSet、TClientDataSet、TDataSetProvider和TDataSource等数据访问组件，再放入TDBNavigator、TDBGrid、两个TBitButton以及一个TRadioGroup组件如图9 5所示。



图9-4 范例InterBase中PRODUCTS数据表的字段和索引信息

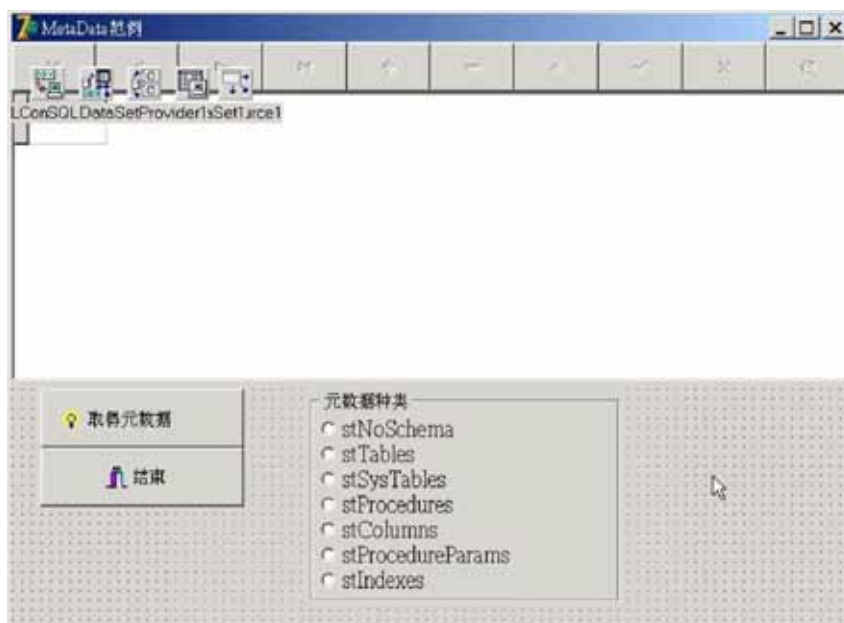


图9-5 范例应用程序的主窗体

主窗体中的TDBGrid用来显示范例程序访问到的元数据。由于元数据是以结果数据集的方式返回的，因此我们可以直接使用数据感知组件来显示元数据。主窗体中的TClientDataSet是用来存储元数据结果数据集的组件，而 TSQLConnection和 TSQLDataSet是用来连接数据库的组件，范例程序分别为这两个组件设置如下的特性值：

TSQLConnection:

特性名称	设置特性值
ConnectionName	D7Books
Name	SQLConnection1

TSQLDataSet:

特性名称	设置特性值
SQLConnection	SQLConnection1
CommandText	select * from PRODUCTS
Name	SQLDataSet1

TDataSource连接到TClientDataSet组件，主窗体中的其他数据感知组件连接到TDataSource。如此一来，当元数据的结果数据集存储到 TClientDataSet之后，数据感知组件就会立刻将元数据显示出来。

最后不要忘记在这个范例程序项目中加入前面实现的 TdbExpressMetaData类。

现在我们就可以开始实现此范例程序以展示客户端如何访问元数据了。首先让我们访问 TSQLDataSet指定的PRODUCTS数据表定义的索引信息，看看我们是否能够像SQL Explorer一样访问到PRODUCTS数据表的索引。下面显示了取得索引信息的实现程序代码：

```
procedure TForm1.DoGetIndexes;
var
  dbMetaData : TdbExpressMetaData;
begin
  dbMetaData := TdbExpressMetaData.Create(Self);

  try
    dbMetaData.SQLConnection := SQLConnection1;
    dbMetaData.MetaDataSet := ClientDataSet1;
    dbMetaData.GetIndexes('PRODUCTS')
  finally
    dbMetaData.Free;
  end;
end;
```

客户端的范例程序首先建立 TdbExpressMetaData对象，然后将主窗体中的 SQLConnection1指定给 TdbExpressMetaData的SQLConnection特性值，让 TdbExpressMetaData使用范例程序建立的数据库连接。再把主窗体中的 TClientDataSet组件指定给 TdbExpressMetaData的MetaDataSet特性值。如此一来，当 TdbExpressMetaData的方法被调用之后包含元数据的结果数据集就会存储在主窗体的 TClientDataSet组件中了。最后，我们调用 TdbExpressMetaData的GetIndexes方法，并且传入要访问索引信息的数据表名称。

在TdbExpressMetaData的GetIndexes方法执行完之后，索引信息的元数据就存储在TClientDataSet中，主窗体的数据感知组件就可以显示这些元数据了，此时我们应该可以看到类似于图9 6的画面，PRODUCTS数据表的索引信息果然呈现在主窗体中。

从这些索引信息我们可以知道 **PRODUCTS** 数据表定义了三个索引对象，其中的主索引是以 **PID** 字段作为索引的目标。



图9-6 范例程序访问数据表索引信息的画面

访问数据库中数据表元数据的方法也和刚才实现的 **DoGetIndexes** 类似，但是通过 **TdbExpressMetaData** 类调用它的 **GetTables** 服务方法，如下所示：

```
procedure TForm1.DoGetTables;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(nil);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetTables(False);
    finally
        dbMetaData.Free;
    end;
end;
```

在执行了 **DoGetTables** 之后，**D7Books** 数据库中所有数据表的信息就呈现在主窗体中了，见图 9 7。

再让我们试着访问 **D7Books** 数据库中存储过程的元数据以及存储过程的参数信

息。图9-8是D7Books数据库中一个添加数据的存储过程 ADDTESTDATA，它接受6个参数并且把传入的数据添加到 PERFTEST数据表中。



图9-7 范例程序访问数据库数据表信息的画面

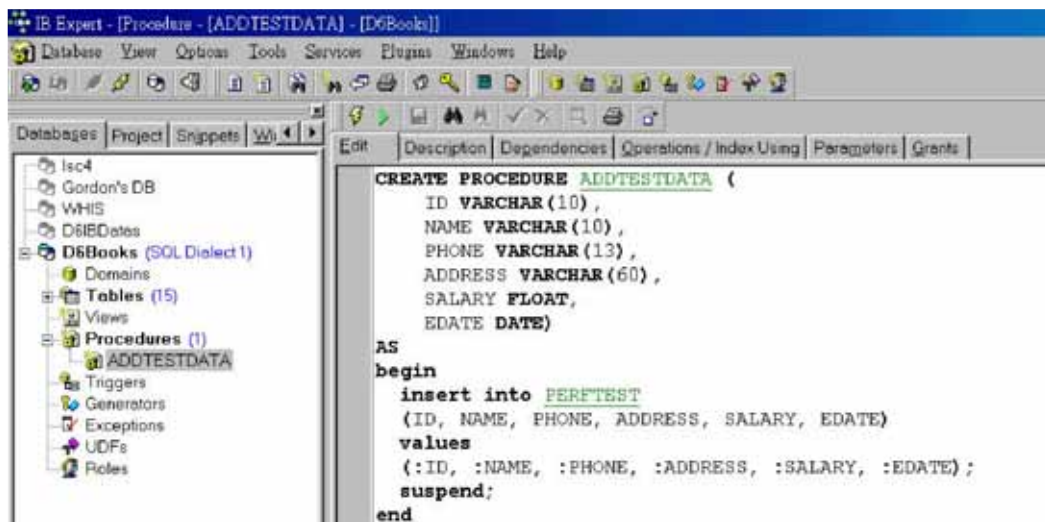


图9-8 数据库中包含参数的存储过程

要取得此存储过程的参数信息元数据也很简单，因为在 TdbExpressMetaData类中就提供了访问特定存储过程的参数元数据的服务方法，客户端调用这个方法就可以取得需要的信息。下面就是客户端实现的程序代码：

```

procedure TForm1.DoGetProcedureParams;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Cr(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetProcedureParameters('ADDTESTDATA');
    finally
        dbMetaData.Free;
    end;
end;

```

上面的程序代码中使用了TdbExpressMetaData类访问存储过程的参数信息，并且在调用TdbExpressMetaData的GetProcedureParameters方法时传入存储过程名称‘ADDTESTDATA’。

现在执行范例程序并且试着访问‘ADDTESTDATA’存储过程的参数信息，我们可以看到类似于图 9-9的画面。从下面的主窗体中我们可以看到存储过程‘ADDTESTDATA’的每一个参数名称、参数类型以及参数次序等详细信息。有了这些存储过程的元数据之后，客户端就可以在应用程序执行时动态地调用这个存储过程。事实上，dbExpress的TSQLStoredProc组件也是使用类似的方式让程序员能够使用对象检视器设置要调用的存储过程以及设置存储过程参数等必要信息。



图9-9 范例程序访问数据库存储过程信息的画面

从上面讨论的内容读者可以知道访问元数据是非常简单、方便的工作，特别是在有了TdbExpressMetaData类的帮助之后。访问元数据让读者开发的应用系统能够动态地调用或访问数据库中的对象，能够执行一些使用静态 dbExpress组件不容易做到的工作。此外，了解元数据也能够帮助用户了解和控制 dbExpress应用系统的性能，在稍后的章节中读者将了解到元数据为什么是非常重要的信息。

下面是此范例程序完整的程序代码，读者可以再试着访问其他种类的元数据并且继续增强此范例程序的功能，甚至让这个范例程序成为处理元数据的通用程序。

```
unit fMetaDataMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DBXpress, FMTBcd, ExtCtrls, DBCtrls, Provider, SqlExpr, DB,
  DBClient, Grids, DBGrids, StdCtrls, Buttons;

type
  TForm1 = class (TForm)
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    ClientDataSet1: TClientDataSet;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
    DataSetProvider1: TDataSetProvider;
    DBNavigator1: TDBNavigator;
    rgMetaDataKind: TRadioGroup;
    bbtnMetaData: TBitBtn;
    bbtnClose: TBitBtn;
    procedure bbtnMetaDataClick(Sender: TObject);
  private
    { Private declarations }
    procedure DoGetNoSchema;
    procedure DoGetTables;
    procedure DoGetSysTables;
    procedure DoGetProcedures;
    procedure DoGetColumns;
    procedure DoGetProcedureParams;
    procedure DoGetIndexes;
  public
    { Public declarations }
  end;
```



```
var
    Form1: TForm1;

implementation

uses uMetaData;

{$R *.dfm}

procedure TForm1.bbtnMetaDataClick(Sender: TObject);
begin
    case rgMetaDataKind.ItemIndex of
        0 : DoGetNoSchema;
        1 : DoGetTables;
        2 : DoGetSysTables;
        3 : DoGetProcedures;
        4 : DoGetColumns;
        5 : DoGetProcedureParams;
        6 : DoGetIndexes;
    end;
end;

procedure TForm1.DoGetColumns;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetColumns('PRODUCTS');
    finally
        dbMetaData.Free;
    end;
end;

procedure TForm1.DoGetIndexes;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);
```

```
try
    dbMetaData.SQLConnection := SQLConnection1;
    dbMetaData.MetaDataSet := ClientDataSet1;
    dbMetaData.GetIndexes ('PRODUCTS');
finally
    dbMetaData.Free;
end;
end;

procedure TForm1.DoGetNoSchema;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetNoSchema;
    finally
        dbMetaData.Free;
    end;
end;

procedure TForm1.DoGetProcedureParams;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetProcedureParameters ('ADDTESTDATA');
    finally
        dbMetaData.Free;
    end;
end;

procedure TForm1.DoGetProcedures;
var
    dbMetaData : TdbExpressMetaData;
```

```
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetProcedures;
    finally
        dbMetaData.Free;
    end;
end;

procedure TForm1.DoGetSysTables;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetSystemTables;
    finally
        dbMetaData.Free;
    end;
end;

procedure TForm1.DoGetTables;
var
    dbMetaData : TdbExpressMetaData;
begin
    dbMetaData := TdbExpressMetaData.Create(Self);

    try
        dbMetaData.SQLConnection := SQLConnection1;
        dbMetaData.MetaDataSet := ClientDataSet1;
        dbMetaData.GetTables(False);
    finally
        dbMetaData.Free;
    end;
end;

end.
```

9.3 观察dbExpress如何使用元数据

从前面的讨论中可以知道，访问数据库的元数据是必要的，因为客户端的数据集组件需要后端数据源的信息才能根据这些元数据组织正确的 SQL 语句来处理数据。元数据影响的不只是 dbExpress 如何选择数据，元数据对于 dbExpress 如何将数据修改回数据源也有重要的影响。在本章最后一个小节中，让我们详细地观察元数据在 dbExpress 中运作的情形。

现在我們希望能够通过 TSQLMonitor 观察 dbExpress 本身使用元数据的情形，以便进一步了解元数据如何影响 dbExpress 的执行行为。要观察元数据在 dbExpress 中的影响，我们可以使用 TSQLMonitor 组件观察在连接数据库、选择数据和更新数据时 dbExpress 是如何处理元数据的，这样就可以有一个完整的概念。

在下面讨论的内容中，我们将使用一个小程序来观察 dbExpress 驱动程序中 MS SQL Server 驱动程序的执行行为。我们将使用 TSQLMonitor 观察客户端 TSimpleDataSet 如何使用 dbExpress 驱动程序来处理数据，并且在这些行动中观察元数据是如何处理的，以此了解元数据对于应用程序的影响。当然笔者也鼓励读者使用相同的方式来观察自己使用的 dbExpress 驱动程序如何处理元数据，例如 dbExpress For Oracle，因为不同的 dbExpress 驱动程序可能会有不同的执行方式。

图9-10是这个小程序执行的画面，当这个小程序开启 MS SQL Server 数据库并且从 pubs 数据库中选择数据时，TSQLMonitor 会记录 dbExpress 在底层使用的所有 SQL 语句和 API。

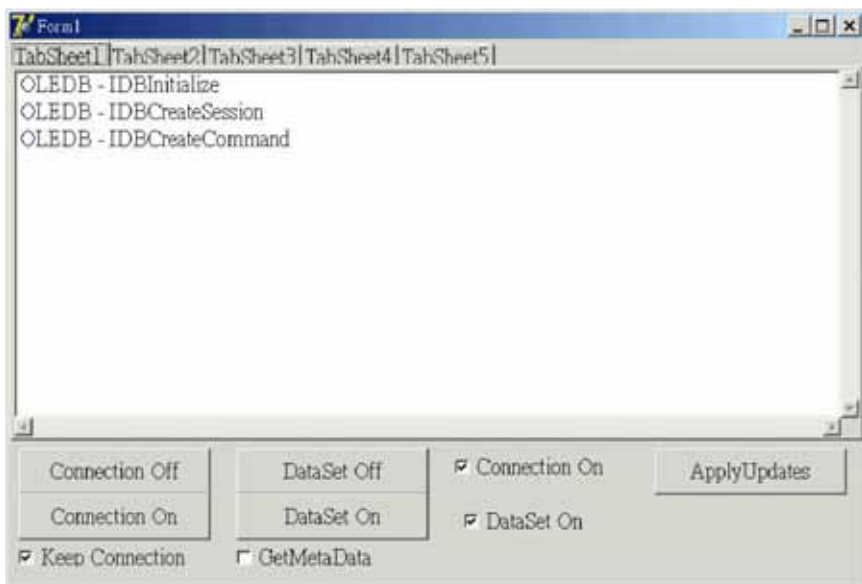


图9-10 观察元数据的范例程序

1. MS Server的起始连接

当范例小程序激活并且开启 TSQLConnection和MS SQL Server的连接时，dbExpress建立了下面的OLE DB接口来初始化连接并且建立与MS SQL Server的连接会话以及一个Command对象准备用来执行SQL命令。

```
OLEDB - IDBInitialize  
OLEDB - IDBCreateSession  
OLEDB - IDBCreateCommand
```

在初始连接状态中，dbExpress的确没有访问任何元数据，因为在这个时期的确没有必要访问元数据，dbExpress不像BDE那样在一开始便会试图访问数据源的元数据，这样设计的好处是启动时的性能会比较好。

2. MS Server选取数据

接着，让我们使用范例小程序开启一个数据表，访问其中的数据以便在数据感知组件中显示，看看dbExpress会如何动作。在我们观察dbExpress的执行行为之前让我们先推想一下，此时由于客户端的 TClientDataSet/TSimpleDataSet组件必须取得后端的结果数据集，因此必须知道要开启的数据表有哪些字段以及字段的类型、长度和个数等的信息，这样才能在客户端的内存中建立内存数据表纲要以存储从结果数据集中取得的数据。因此此时dbExpress一定会访问元数据以向客户端 TClientDataSet/TSimpleDataSet组件提供需要的信息。下面就是当程序开启数据表时dbExpress执行的OLE DB命令：

```
01. OLEDB - ICommandText  
02. select * from employee  
03. OLEDB - SetCommandText  
04. OLEDB - ICommandPrepare  
05. OLEDB - Prepare  
06. OLEDB - Execute  
07. OLEDB - GetResult  
08. OLEDB - IColumnsInfo  
09. OLEDB - GetColumnInfo  
10. OLEDB - Release  
11. OLEDB - IAccessor  
12. OLEDB - CreateAccessor  
13. OLEDB - GetNextRows  
14. OLEDB - GetData  
15. OLEDB - ReleaseRows  
16. ...  
17. OLEDB - Release  
18. OLEDB - ReleaseAccessor
```

下面的表格详细说明了上面OLE DB命令的意义：

行 号	说 明
01	dbExpress建立OLE DB中的 ICommandText接口以准备使用SQL语句
02	dbExpress将客户端的SQL语句传递给后端的MS SQL Server
03	dbExpress在 ICommandText中指定要使用的SQL语句
04	dbExpress建立OLE DB中的 ICommandPrepare接口以编译SQL语句, 准备执行
05	要求MS SQL Server编译准备要执行的SQL语句
06	要求MS SQL Server执行编译的SQL语句
07	取得执行SQL语句产生的结果数据集
08	dbExpress建立OLE DB中的 IColumnsInfo接口以准备取得数据表的元数据
09	从MS SQL Server取得元数据
10	释放 IColumnsInfo接口
11	dbExpress声明OLE DB中的 IAccessor接口以准备访问结果数据集
12	建立 IAccessor接口COM对象
13	使用 IAccessor接口取得下一笔可供访问的数据
14	从MS SQL Server取得 IAccessor指向的数据
15	释放已取得的数据目前的 IRowSet接口
16	重复步骤14和15以取得所有数据
17	释放 IColumnsInfo接口
18	释放 IAccessor接口

从上面表格中的步骤 8, 9中我们可以证明, dbExpress的确如我们所料在取得了结果数据集之后立刻访问数据表的字段等元数据以便让客户端的 TClientDataSet/TSimpleDataSet组件建立内存数据表结构, 此时 dbExpress开始执行访问元数据的操作, 而这是程序员使用 dbExpress开启数据表时必须连带付出的代价。

3. MS Server更新数据

观察了dbExpress如何选取数据之后, 再让我们看看在修改数据时 dbExpress是否需要访问元数据。图 9 11显示了我们继续在范例小程序中更新数据, 最后使用范例

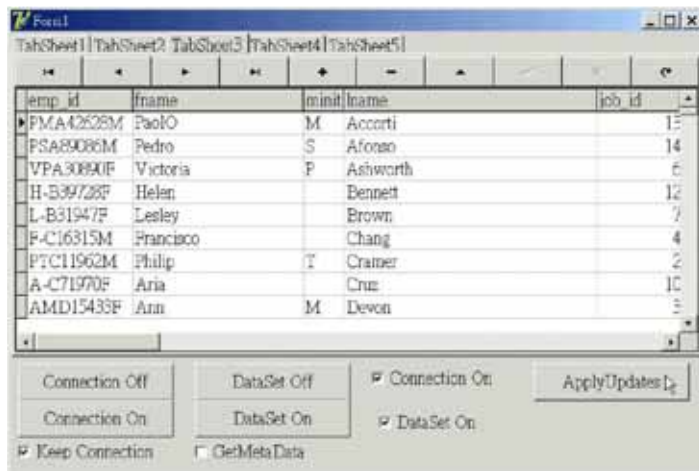


图9-11 使用范例程序把数据更新回 MS SQL Server 2000

程序窗体中的ApplyUpdates按钮把数据更新回MS SQL Server中。

下面的内容是范例程序通过 TSQLMonitor观察到dbExpress执行的动作：

```
01. OLEDB - ITransactionLocal
02. OLEDB - StartTransaction
03. OLEDB - ICommandText
04. update "employee" set
    "fname" = ?
where
    "emp_id" = ? and
    "fname" = ? and
    "minit" = ? and
    "lname" = ? and
    "job_id" = ? and
    "job_lvl" = ? and
    "pub_id" = ? and
    "hire_date" = ?

05. OLEDB - SetCommandText
06. OLEDB - ICommandPrepare
07. OLEDB - Prepare
08. OLEDB - IAccessor
09. OLEDB - CreateAccessor
10. OLEDB - Execute
11. OLEDB - GetResult
12. OLEDB - ReleaseAccessor
13. OLEDB - Release
14. OLEDB - Release
15. OLEDB - Release
16. OLEDB - Commit
```

让我们解释一下在修改数据时 dbExpress执行了什么工作？

行 号	说 明
01	建立OLE DB的ITransactionLocal接口，准备使用MS SQL Server的事务管理
02	激活事务
03	建立ICommandText接口以准备执行SQL语句
04	传递要执行的SQL语句
05	设置要执行的SQL语句
06	dbExpress建立OLE DB中的ICommandPrepare接口以编译SQL语句，准备执行
07	要求MS SQL Server编译准备要执行的SQL语句
08	dbExpress声明OLE DB中的IAccessor接口以准备访问结果数据集
09	建立IAccessor接口COM对象
10	要求MS SQL Server执行编译的SQL语句

(续)

行 号	说 明
11	取得结果数据集
12	释放IAccessor接口
13	释放代表结果数据集的接口
14	释放ICommandPrepare接口
15	释放ICommandText接口
16	确定在事务中执行SQL语句

从上面的分析我们可以得到下面的观察结果：

- 在dbExpress更新数据时似乎没有使用元数据，这是真的吗？
- 当dbExpress使用SQL语句进行更新时，目前的 dbExpress驱动程序似乎执行了数项多余的工作。例如，在这个行动中我们执行 `update` 指令以更新数据。而 `update` 不会返回结果数据集，那么为什么 dbExpress 仍然会执行上面的步骤8、9、11和12呢？这不是不必要的吗？基本上，dbExpress 执行步骤8、9、11和12是为了了解这个 `update` 命令影响了后端的多少个记录以及了解这个 `update` SQL 语句是否成功地执行完毕以便执行异常处理的工作，因此并不算是多余的工作。

现在让我们使用 MS SQL Server 的 Profiler 来回答前面的第一个问题，看看在更新数据的行动中有没有使用元数据。图 9-12 显示了 dbExpress 在进行更新时向 MS SQL Server 下达的命令。

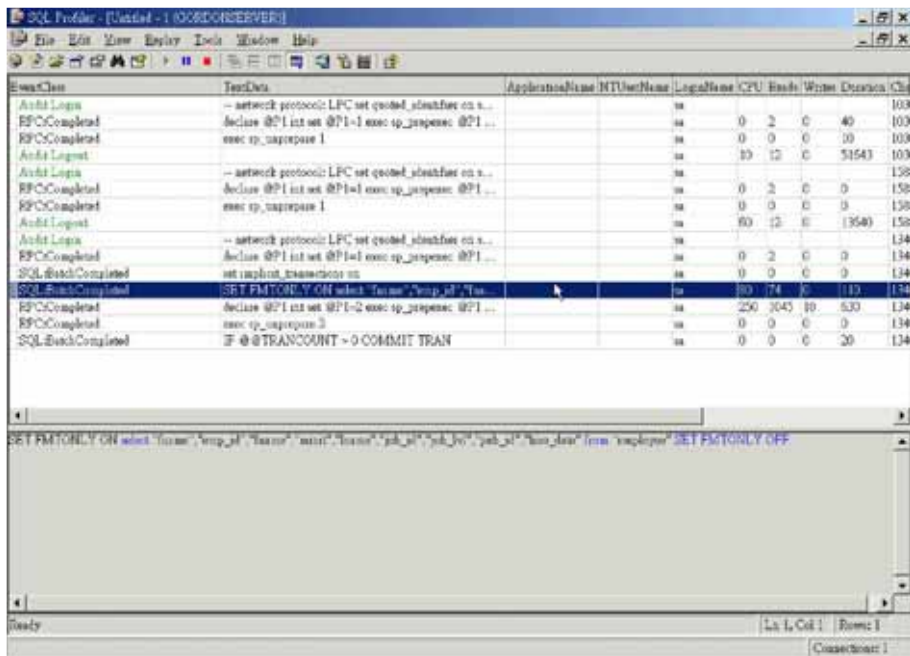
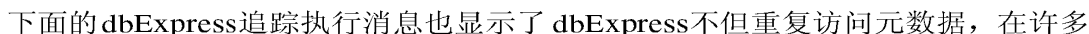


图9-12 MS SQL Server的Profiler清楚地显示了在dbExpress更新数据之前的确使用了元数据



工作方面也重复执行，例如在 **CreateAccessor** 方面dbExpress应该可以在所有更新动作之后再一次取得所有受影响的记录数，应该不需要重复建立 OLE DB对象，再释放 OLE DB对象。

```
OLEDB - ITransactionLocal
OLEDB - StartTransaction
OLEDB - ICommandText
update "employee" set
    "fname" = ?
where
    "emp_id" = ? and
    "fname" = ? and
    "minit" = ? and
    "lname" = ? and
    "job_id" = ? and
    "job_lvl" = ? and
    "pub_id" = ? and
    "hire_date" = ?
```

```
OLEDB - SetCommandText
OLEDB - ICommandPrepare
OLEDB - Prepare
OLEDB - IAccessor
OLEDB - CreateAccessor
OLEDB - Execute
OLEDB - GetResult
OLEDB - ReleaseAccessor
OLEDB - Release
OLEDB - Release
OLEDB - Release
OLEDB - ICommandText
update "employee" set
    "fname" = ?
where
    "emp_id" = ? and
    "fname" = ? and
    "minit" = ? and
    "lname" = ? and
    "job_id" = ? and
    "job_lvl" = ? and
    "pub_id" = ? and
    "hire_date" = ?
```

```
OLEDB - SetCommandText
OLEDB - ICommandPrepare
OLEDB - Prepare
OLEDB - IAccessor
OLEDB - CreateAccessor
OLEDB - Execute
OLEDB - GetResult
OLEDB - ReleaseAccessor
OLEDB - Release
OLEDB - Release
OLEDB - Release
OLEDB - ICommandText
update "employee" set
    "fname" = ?
where
    "emp_id" = ? and
    "fname" = ? and
    "minit" = ? and
    "lname" = ? and
    "job_id" = ? and
    "job_lvl" = ? and
    "pub_id" = ? and
    "hire_date" = ?
```

```
OLEDB - SetCommandText
OLEDB - ICommandPrepare
OLEDB - Prepare
OLEDB - IAccessor
OLEDB - CreateAccessor
OLEDB - Execute
OLEDB - GetResult
OLEDB - ReleaseAccessor
OLEDB - Release
OLEDB - Release
OLEDB - Release
OLEDB - Commit
```

从这些观察中我们可以知道，dbExpress在处理修改数据时是使用最保守且安全的方式，但不是最有效率的方式。这种方式在客户端修改少量数据的情形下没有什么大的影响，读者可以直接接受dbExpress处理的方式。但是如果读者需要在客户端大量处理修改的数据，那么最好在客户端自己使用TSQLDataSet通过SQL语句来直接处理，避免使用TClientDataSet/TSimpleDataSet的ApplyUpdates方法。

4. 访问MS Server的元数据

在离开本小节之前，让我们再观察一下如果客户端应用程序直接使用 dbExpress 组件提供的访问元数据功能，例如使用 TSQLConnection组件的GetTableNames方法访问数据库中的所有数据表名称，那么 dbExpress驱动程序会如何完成工作。下面的程序代码是我们在范例小程序中加入的取得 pubs数据库中数据表名称的事件处理函数：

```
procedure TForm1.btnGetTablesClick(Sender: TObject);
var
    aList : TStringList;
begin
    mmMemo := Memo4;
    aList := TStringList.Create;
    try
        SQLConnection1.GetTableNames(aList, False;
    finally
        FreeAndNil(aList);
    end;
end;
```

当我们在范例小程序中执行上面的程序代码时，可以观察到 MS SQL Server的 dbExpress驱动程序执行下面的 OLE DB命令：

```
OLEDB - ICommandText
OLEDB - IDBSchemaRowset
OLEDB - GetRowset
OLEDB - IColumnsInfo
OLEDB - GetColumnInfo
OLEDB - Release
OLEDB - IAccessor
OLEDB - CreateAccessor
OLEDB - GetNextRows
OLEDB - Release
OLEDB - ReleaseAccessor
OLEDB - Release
```

请注意在上面的 OLE DB命令中，当客户端应用程序访问元数据时，MS的 dbExpress驱动程序会取得 OLE DB的IDBSchemaRowset接口以取得元数据，再建立 IColumnsInfo接口以取得返回的元数据本身的描述信息。不过更重要的是，此时 MS的Profiler显示了当客户端访问元数据信息时，dbExpress驱动程序会使用一个额外的独立线程连接MS SQL Server以取得元数据信息，等到取得元数据之后，这个额外的线程会立刻被释放。我们可以从图 9 14所示的Profiler中看到一个线程连接进行 Auto Login，在访问元数据之后立刻执行 Auto Logout释放连接。

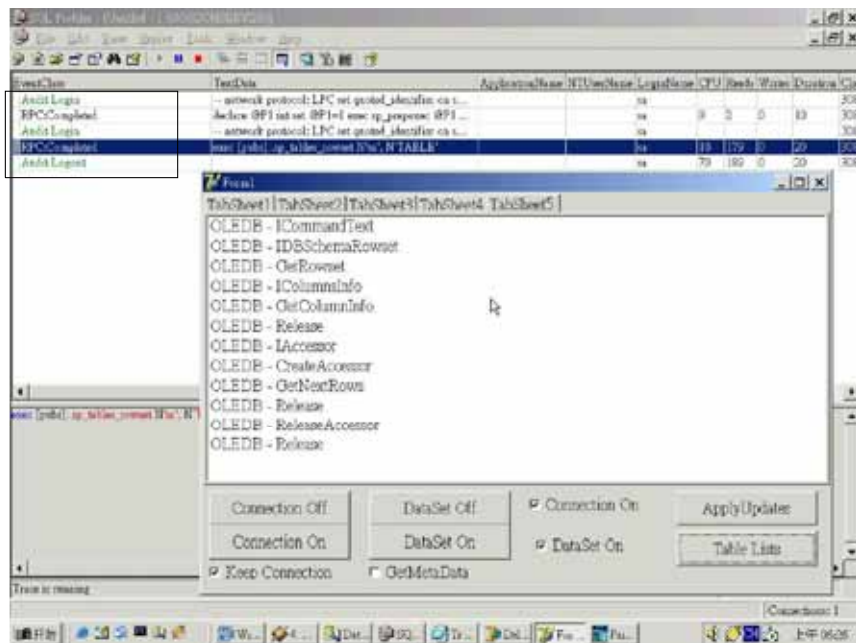


图9-14 在客户端直接访问元数据时，dbExpress会使用额外的线程

由于dbExpress使用了额外的连接线程为客户端访问元数据，所以会暂时增加客户端的数据库连接数量，因此如果读者的客户端应用程序使用的数据库有连接数量的限制，那么就必须注明这些会增加客户端连接次数的动作，否则可能会发生客户端应用程序不定时被拒绝连接的情形。

不过不管读者选择使用什么方式，我们从前面的讨论中都可以证实元数据在dbExpress处理数据时占有非常重要的地位。

从上面的观察中我们也可以得到以下的重要结论：

- 不同的dbExpress驱动程序可能会使用不同的方式来处理元数据。
- 目前的dbExpress驱动程序似乎在连接时不会先取得元数据，这会增加启动效率。
- 虽然在dbExpress的联机帮助中说明当TSQLConnection的KeepConnection被设置为False时，如果没有任何DataSet是开启的，那么TSQLConnection会自动关闭数据库连接以节省资源。不过从现在实现的dbExpress驱动程序来看，似乎并没有100%地实现这种方式。不管KeepConnection是True还是False，TSQLConnection并不会关闭它的主要连接。
- TSimpleDataSet的DataSet\GetMetadata可以控制客户端数据集是否先取得数据表的元数据。不过目前的dbExpress驱动程序似乎也没有100%地遵守此规则。不管GetMetadata是True还是False，目前的dbExpress似乎都尽量避免事先取得

元数据以增加性能。只有在把数据修改回数据源时才取得必要的元数据。

- 开发人员在使用 dbExpress 驱动程序时最好先使用工具观察一下 dbExpress 驱动程序的重要执行行为。
- 在 WAN、Internet/Intranet 或缓慢的网络环境中，开发人员最好避免访问不必要的元数据以增加性能。例如在 WAN/Internet/Intranet/缓慢网络环境中不管使用的驱动程序为何，对于只读的数据访问一定要记得把 `GetMetadata` 设置为 `False`，同时在这种应用中如果需要经常更新数据，那么也要记得不要使用 `TClientDataSet/TSimpleDataSet` 的 `Applyupdates` 方法，最好是使用额外的 `TSQLDataSet/TDataSetProvider`，设置 `TDataSetProvider` 的 `ResolveToDataSet` 为 `True` 并且直接使用 SQL 语句搭配 Delphi 程序代码来更新数据，以避免 `ApplyUpdates` 每一次修改数据时都需要向数据源取得元数据，这样可以大幅增加性能。
- dbExpress 驱动程序会使用额外的连接线程为客户端访问元数据，因此客户端应该尽可能避免不必要的元数据访问动作。

记住，尽量避免不必要的数据和元数据访问可以让你的 dbExpress 应用程序执行得更为迅速，这是不变的法则。

9.4 结论

在本章中我们详细讨论了元数据，为什么元数据很重要？如何使用 dbExpress 访问元数据？最后我们还使用小工具来观察 dbExpress 本身如何处理元数据，让我们进一步了解元数据在 dbExpress 中运作的情形。

一般的 Delphi/Kylix 的数据库书籍很少讨论元数据，要么是因为这些书籍的作者不甚了解元数据，要么是因为他们认为元数据不重要或是属于比较底层的东西因此并没有多加讨论。不过通过本章的讨论读者现在应该知道元数据是很重要的，因为它们对于 dbExpress 的执行行为有非常大的影响。特别是当开发人员在各种不同的应用中使用 dbExpress 时，元数据对于应用系统就更重要了，因为不适当地使用 dbExpress 和控制元数据会让开发人员的 dbExpress 应用系统在不同的应用中呈现不同的反应，有时很快，有时似乎又非常缓慢，而失去了应用程序一致的品质。

本章的内容应该会让 Delphi/Kylix 的开发人员了解元数据，即使开发人员仍然不愿意或是不需要在应用系统中使用元数据，建立对于元数据的正确认识仍然可以帮助开发人员在必要时解决有关性能或是元数据方面的问题和 bug。

第五部分

性能篇



第10章 开发高效率的数据库应用系统

不管使用什么开发工具或是什么数据访问引擎，笔者认为所有程序员大概都想要开发性能良好的应用程序，而能不能编写出比别的程序员的程序执行得更快、更稳定的应用程序也就成了评价一个程序员好坏的标准之一。

性能对于数据库应用系统来说更是重要，尤其是当应用程序需要处理大量数据时，一个编写良好的数据库应用程序会比差劲的程序有着数倍，甚至是数十到数百倍的性能差距。这种差距可能意味着好的数据库程序只需要执行数秒，而差劲的程序则需要执行数 10 分钟。但是为什么使用相同工具的应用程序会有这么大的差距呢？

当然，问题的答案就是程序员是否能够好好地掌握他 / 她使用的工具。在前面的章节中，本书已经解释过 Delphi / Kylix 的 dbExpress 引擎使用了单向的游标访问数据，而且 dbExpress 本身没有内建任何缓冲区机制，因此可以算是速度最快的数据访问引擎之一了。因此读者可能认为这样一来用 Delphi / Kylix 的 dbExpress 开发的数据库应用程序执行得就非常快速了，应该没有什么地方可以再调整性能了，因为 dbExpress 引擎似乎不像 BDE / IDAPI 那样可以使用 Delphi 的 SQL Explorer 来调整任何参数机制。

没错，使用 Delphi / Kylix 和 dbExpress 开发出来的数据库应用程序的确已经执行得很有效率了，如果读者认为已经没有任何调整 dbExpress 处理数据速度的方法，那么读者也没有错，因为这是大多数人的想法。不过聪明的程序员仍然有办法从 dbExpress 中压榨出更多的性能，在这一章中笔者就会告诉读者这些聪明的程序员是如何做的，让读者也能够和这些聪明的程序员一样。本章讨论的内容也是本书的精华之一，是读者在其他书籍中看不到的，如果读者在其他 Delphi 书籍中看到和本章类似的内容，那么读者可以知道这些内容是从那里参考来的，呵呵。

10.1 从测试 dbExpress、BDE / DAP 和 dbExpress 开始

想要调整 dbExpress 的性能，程序员就必须深入地了解 dbExpress 是如何处理数据的。除了了解 dbExpress 的执行行为之外，让我们一起来看看 BDE / IDAPI 和 dbExpress 比较起来到底谁的速度比较快，毕竟 Delphi / Kylix 想使用 dbExpress 取代 BDE / IDAPI，而 BDE / IDAPI 是已经开发了数年的数据访问引擎，在性能上有一定的水准，因此

dbExpress必须至少拥有和BDE/IDAPI类似的性能才可以。而在我们了解了 dbExpress 的执行行为之后，如果调整过的 dbExpress性能能够超过 BDE/IDAPI的性能，那么就太好了，不是吗？

图10 1是本章使用的性能测试程序的主窗体，在这个测试程序中将使用 dbExpress、BDE/IDAPI和IBExpress三种不同的数据引擎对 InterBase数据库处理数据。这个测试程序将比较三种引擎处理少量、中量到大量数据的执行行为，同时也分析三种引擎对于不同的数据量以及不同的数据访问行为的反应。



图10-1 性能测试程序的主窗体

笔者使用的InterBase是FireBird 1.x的版本。FireBird是InterBase的开放源代码版本之一，它可以同时支持 InterBase 5.x和InterBase 6.x的数据库格式，读者可以在WWW.FireBird.com 处下载这个数据库，当然读者也可以使用其他数据库来进行测试。在使用 dbExpress测试不同的数据库时，每一个 dbExpress驱动程序的执行结果可能会和本章讨论的结果不同，这应该是每一个 dbExpress驱动程序实现的品质以及不同的数据库服务器本身的差异造成的。读者在了解了本章讨论的内容之后可以自己进行实际的测试以及性能调整。

图10 2是使用三种不同引擎的数据模块。对于这三种不同的访问引擎，本章都使用了它们默认的状态， dbExpress使用了TSQLConnection和TSQLClientDataSet，BDE/IDAPI使用了RequestLive设置为True的TQuery组件，以便让BDE/IDAPI引擎来处理所有数据访问工作，而不加入任何人工的调整。而 IBExpress使用了其预设的最

快速的数据访问方式，IBDatabase加IBTransaction加IBQuery和IBUpdateSQL，以便让dbExpress和原始的InterBase访问组件进行比较。

下表是测试程序对于不同引擎使用的测试结构。为了测试 dbExpress和BDE/IDAPI真正的性能差距，因此在 BDE/IDAPI的结构中本章使用了 BDE/IDAPI的主从结构，这样可以测试出当 dbExpress面对原始的主从结构访问引擎时，在性能上是否能够匹配。

数据引擎	测试结构
dbExpress	dbExpress组件+DataSnap
BDE/IDAPI	主从结构
IBExpress	IBExpress组件+DataSnap

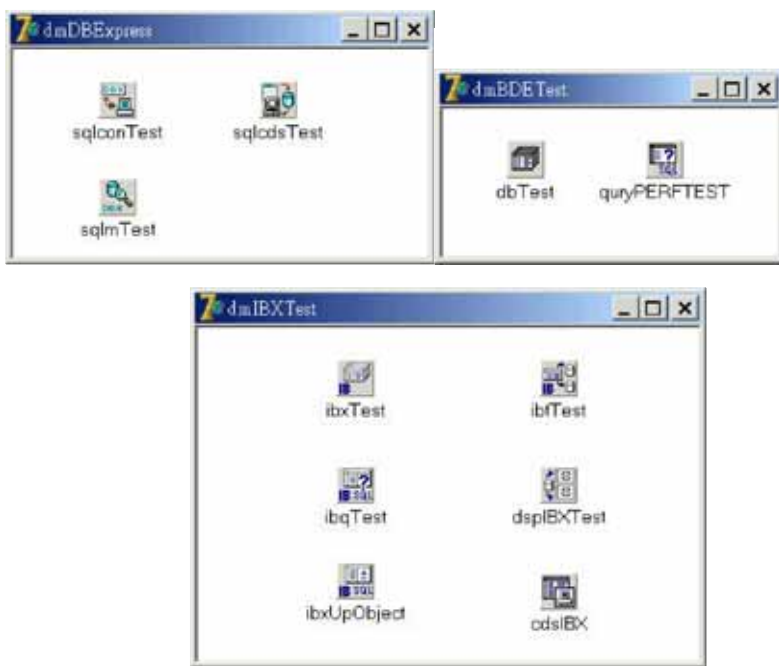


图10-2 测试程序使用的三种数据引擎的组件和数据模块

首先，测试程序先使用 dbExpress连接并且开启空白的数据库，以测试 dbExpress的初始化连接速度。图 10 3是dbExpress执行的结果画面。从dbExpress执行的结果可以看到，它是以正常的速度执行数据库连接初始化和开启数据表。dbExpress的这项执行速度算是非常令人满意，因为它比 BDE/IDAPI表现得还好，这可以证明 dbExpress只使用单向游标的方式处理数据的确是非常快速。

现在，让我们观察一下 dbExpress在连接和开启空白的数据表时到底执行了哪些工作。



图10-3 测试dbExpress连接和开启数据库的速度

10.1.1 观察dbExpress的执行行为之一

下面的列表是当dbExpress连接并且访问空白的InterBase数据表时向InterBase发出的SQL命令。从这个列表中我们可以整理出dbExpress的重要行为。

```

INTERBASE - isc_dsqli_allocate_statement
INTERBASE - isc_start_transaction
Select * from PERFTEST
INTERBASE - isc_dsqli_prepare
INTERBASE - isc_dsqli_describe_bind
INTERBASE - isc_dsqli_execute
INTERBASE - isc_dsqli_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, B.RDB$INDEX_NAME, RDB$FIELD_NAME,
B.RDB$FIELD_POSITION, A.RDB$INDEX_TYPE, A.RDB$UNIQUE_FLAG,
C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE FROM RDB$INDICES A, RDB$INDEX_SEGMENTS
FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C ON A.RDB$RELATION_NAME =
C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY' WHERE
(A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG = 1) AND (A.RDB$INDEX_NAME = B.RDB$
$INDEX_NAME) AND (A.RDB$RELATION_NAME = 'PERFTEST') ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsqli_prepare
INTERBASE - isc_dsqli_describe_bind
INTERBASE - isc_dsqli_execute
INTERBASE - isc_dsqli_fetch
INTERBASE - isc_dsqli_fetch

```

访问数据库的元数据

访问空白数据表 PERFTEST

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
```

- 每当dbExpress执行数据库访问动作时，都会先激活一个数据库事务。
- 当dbExpress连接数据库时，会从数据库中取得数据库和数据表的元数据。
- 当dbExpress访问数据表的数据时，会先要求InterBase编译访问数据的SQL语句，再执行此SQL语句（使用Prepared Query）。
- 最后当数据访问结束之后，dbExpress会释放数据库编译的SQL语句，并且Commit激活的数据库事务。

请读者注意，随着dbExpress版本的不断进步，读者使用的dbExpress的追踪结果可能和本书列出的结果不完全一致。

仅仅从一个数据访问的行为我们无法确实了解dbExpress的执行行为，现在让我们继续使用dbExpress添加测试程序随机产生的数据，看看dbExpress如何把客户端应用程序的数据修改到数据库中。

图10-4是在测试数据表中使用dbExpress添加两个随机记录的执行结果，从图中可以看到dbExpress在添加少量数据方面执行的速度也非常快。



图10-4 dbExpress添加随机产生的数据的速度

10 1 2 观察dbExpress的执行行为之二

现在再让我们观察当 dbExpress添加数据时的执行行为，下面的列表是图 10 4 执行的结果：

```

INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
Select * from PERFTEST
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, RDB$INDEX_NAME, RDB$FIELD_NAME,
B.RDB$FIELD_POSITION, A.RDB$INDEX_TYPE, EA.RDB$UNIQUE_FLAG,
C.RDB$CONSTRAINT_NAME, B.CONSTRAINT FROM RDB$INDICES
RDB$INDEX_SEGMENTS, L OUTER JOIN RDB$RELATION_CONSTRAINTS
A.RDB$RELATION_NAME C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE 'PRIMARY
KEY' WHERE (A.RDB$SYSTEM_FLAG > A.RDB$SYSTEM_FLAG) AND
(A.RDB$INDEX_NAME = RDB$INDEX_NAME) AND (A.RDB$RELATION_NAME = RDB$RELATION_NAME)
('PERFTEST')) ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
insert into PERFTEST
( ID , NAME , PHONE , ADDRESS , SALARY , BDATE
values
(?, ?, ?, ?, ?, ?))

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
insert into PERFTEST
( ID , NAME , PHONE , ADDRESS , SALARY , BDATE
values

```

(?, ?, ?, ?, ?,)?

```
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_retaining
```

从上面的列表中我们看到了 dbExpress 黑暗的一面。当 dbExpress 修改数据时，它似乎一定会执行下面的动作：

- 激活数据库事务。
- 访问数据库和数据表的元数据。
- 为每一个修改的记录动态产生 SQL 语句，并且要求数据库产生一个编译的 SQL，执行这个编译的 SQL，最后再释放这个编译的 SQL。
- 最后 Commit 数据库事务。

从上面的列表和分析中我们可以看到 dbExpress 没有效率的地方（严格地说，应该是 dbExpress 组件 + DataSnap，而不是 dbExpress 引擎本身），那就是 DataSnap 会为每一个修改的记录动态产生 SQL 语句，再通过 dbExpress 要求数据库引擎执行这个 SQL 语句。这对于修改较少数据的应用程序而言并不会产生什么问题，但是对于需要处理大量修改数据的应用程序而言，就需要付出非常大的代价。

要解决 DataSnap + dbExpress 在处理大量数据时比较低效率的缺点，程序员就必须让 DataSnap 在修改数据时使用比较聪明的方法，那就是当 Delphi 的应用程序修改数据时，如果这些需要修改的数据都要修改到相同的数据表而且只有字段值不同，那么我们就希望 DataSnap 能够重复使用已经产生的 SQL 语句。最好数据库引擎也能够保存并且继续使用已经编译好的 SQL 语句，而不需要重复编译相同的 SQL 语句。

在稍后的章节中，本书会说明如何让 DataSnap 能够满足我们这样的需求。在说明这个方法之前，让我们继续观察 dbExpress 和 BDE/IDAPI 的执行行为，以便了解如何进一步优化 dbExpress。

图 10 5 和图 10 6 是使用 BDE/IDAPI 连接相同的数据库、访问相同的数据表以及添加数据的执行结果，读者可以比较 dbExpress 和 BDE/IDAPI 在连接和访问数据时需要花费的时间。

从上面的图形中我们似乎可以观察到 dbExpress 在连接数据库方面比较快，但是在处理数据方面似乎比 BDE/IDAPI 慢，本书会在稍后说明出现这个现象的原因。在此之前，让我们观察 dbExpress 和 BDE/IDAPI 在处理大量数据时的执行行为。

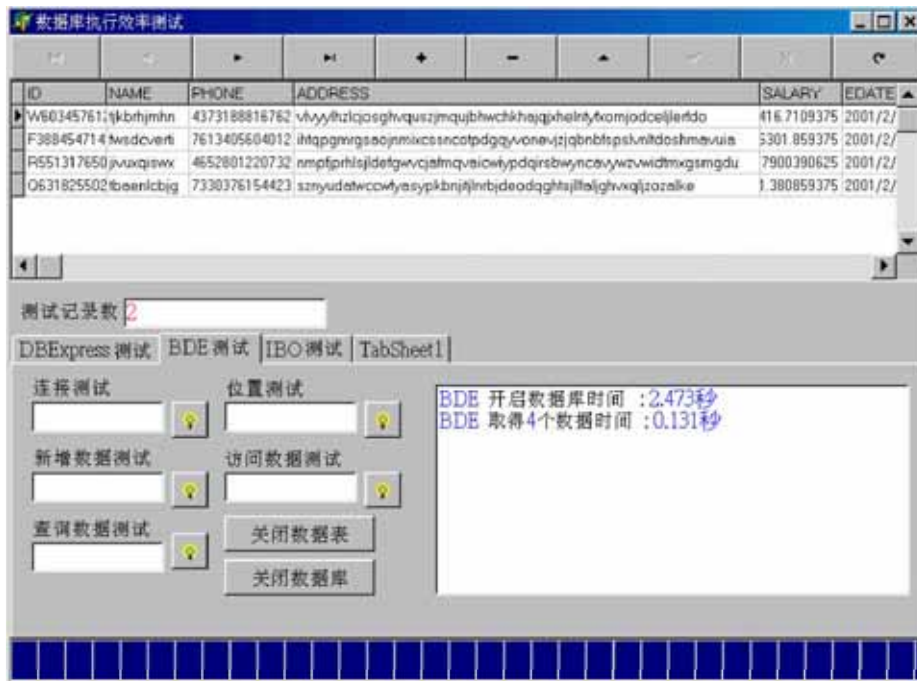


图10-5 BDE/IDAPI连接InterBase的执行结果

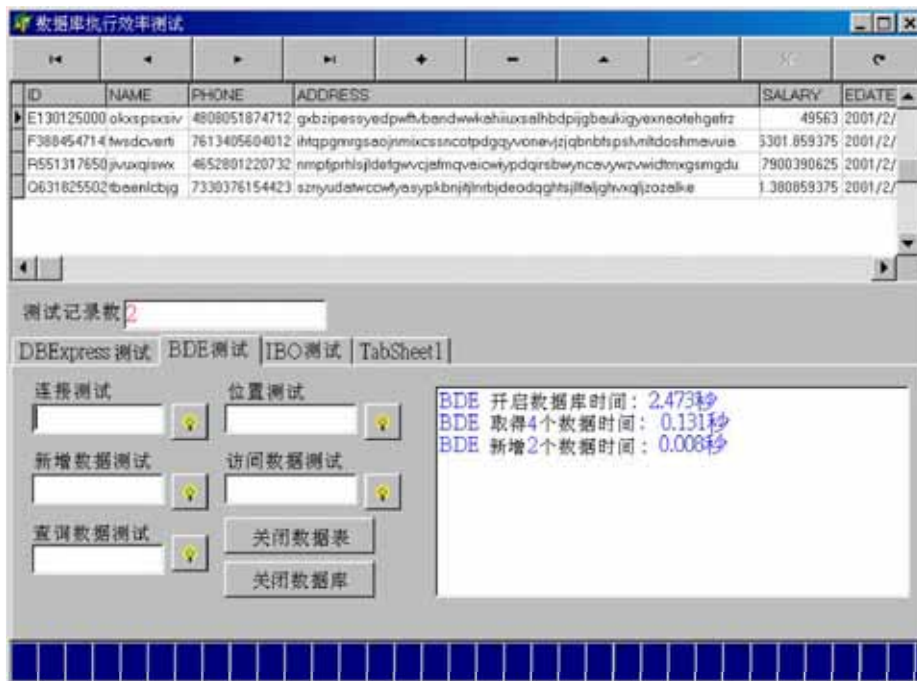


图10-6 BDE/IDAPI添加随机产生的数据的速度

图10 7和图10 8分别是dbExpress和BDE/IDAPI在处理1000个记录时花费的时间,从这两个结果我们似乎观察到BDE/IDAPI比dbExpress有效率,难道真是姜是老的辣,dbExpress在处理大量数据时就比不上BDE/IDAPI吗?在我们下结论之前请再看看下面的两个访问大量数据的情形(图10 9和图10 10)。

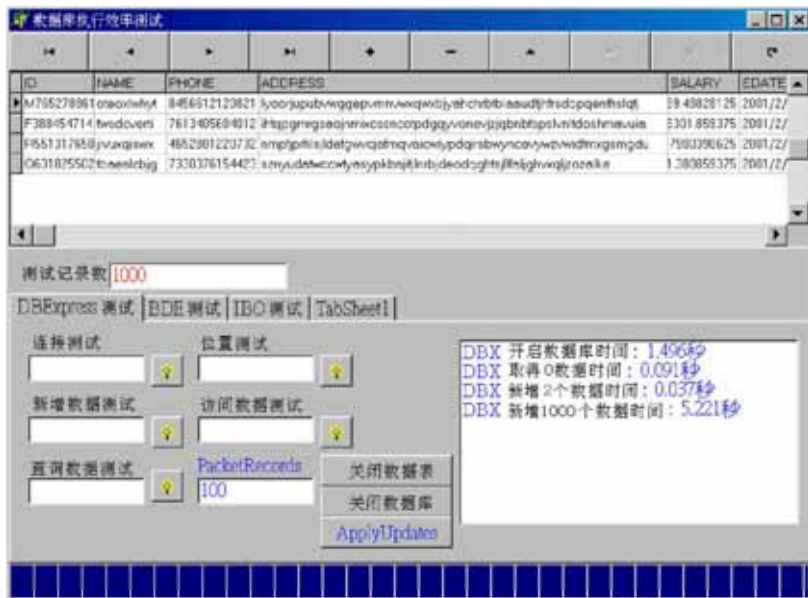


图10-7 dbExpress添加大量随机产生的数据的速度

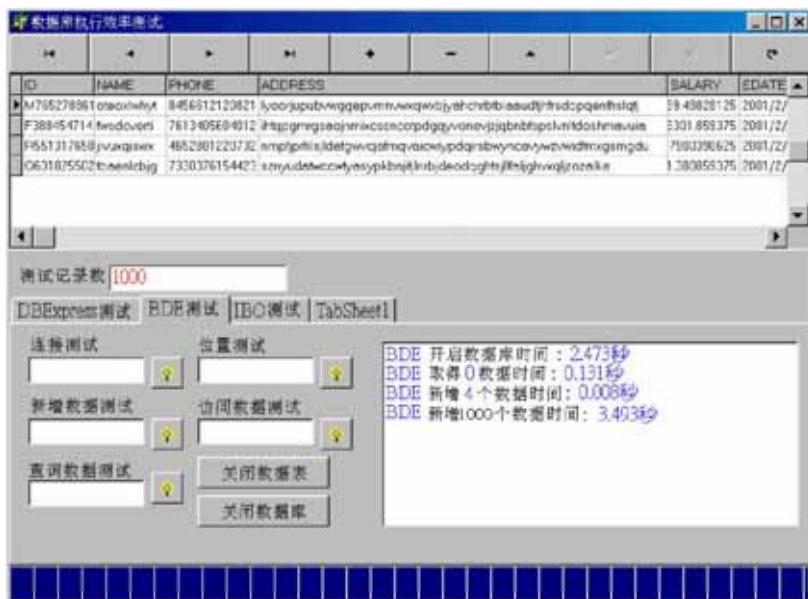


图10-8 BDE/IDAPI添加大量随机产生的数据的速度

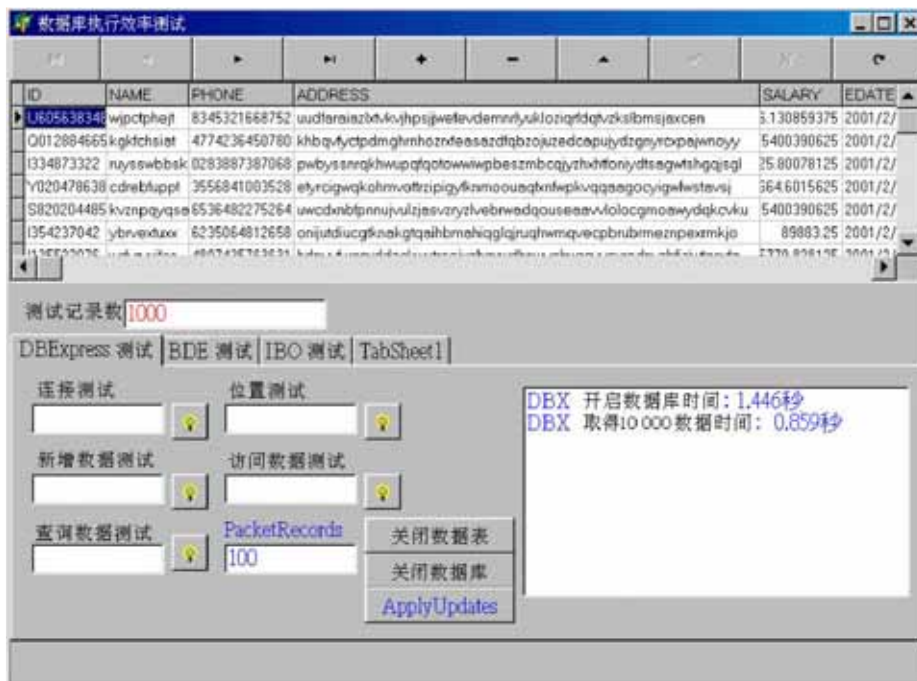


图10-9 dbExpress访问大量数据的速度

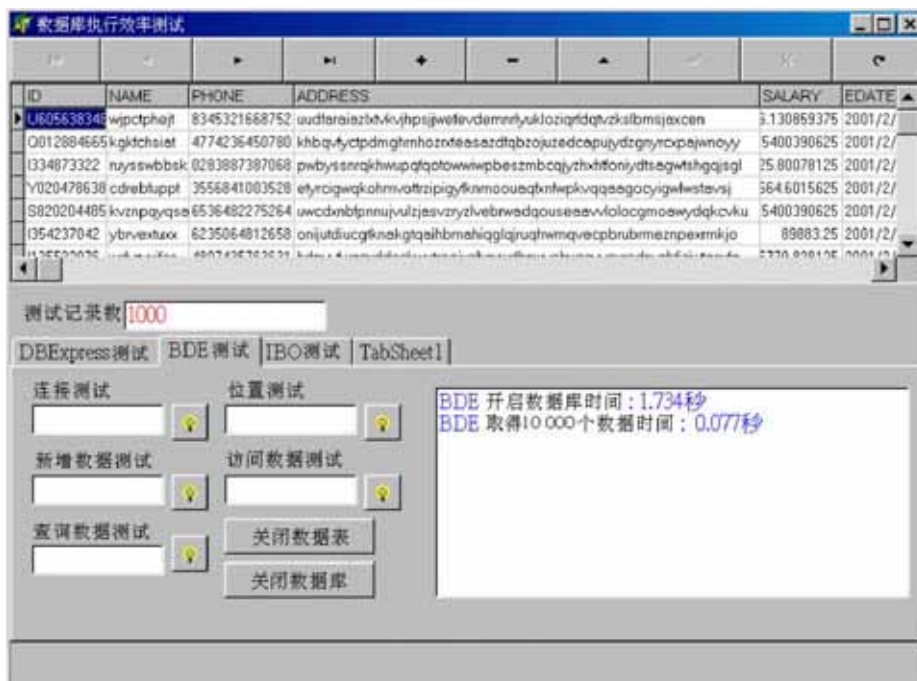


图10-10 BDE/IDAPI访问大量数据的速度

这两个执行画面分别是 dbExpress 和 BDE/IDAPI 从数据表中取出 1000 个记录的结果，从这两个结果中我们再度看到 BDE/IDAPI 以 10 多倍的速度超过了 dbExpress，难道 dbExpress 只是适合处理少量数据的数据引擎吗？

10.2 dbExpress 和 BDE/IDAPI 的性能比较

经过了前一小节的讨论和观察，我们发现了 dbExpress 的许多执行行为，但是也产生了不少的困惑，因为在许多处理数据的应用方面 dbExpress 似乎都比 BDE/IDAPI 缓慢，甚至在某些方面 dbExpress 还比 BDE/IDAPI 缓慢了 10 倍之多，这是不是说明了我们还是应该继续使用 BDE/IDAPI？因为 BDE/IDAPI 毕竟开发了数年之久，比新的 dbExpress 引擎历史更久，因此 BDE/IDAPI 比 dbExpress 更有效率，不是吗？

当然不，前面观察到的结果只能说明 dbExpress 和 BDE/IDAPI 的执行行为不同，并不能说明 dbExpress 的性能比不上 BDE/IDAPI，事实上在没有经过调整之前 dbExpress 的执行速度已经和 BDE/IDAPI 不相上下了，调整之后的 dbExpress 更有不凡的性能。如果读者在看过前面的章节之后，仍然无法解释为什么看起来 BDE/IDAPI 比 dbExpress 更有效率，那么就代表读者并不真正了解 BDE/IDAPI 和 dbExpress。

在下面的三个小节中，本书将以三种不同的数据库动作来比较和说明 dbExpress 和 BDE/IDAPI 的执行行为和性能。相信读者在看完这个三小节之后，便会真正了解 BDE/IDAPI 和 dbExpress 的异同了。

10.2.1 连接数据库的速度

下面的表格和图 10-11 是 dbExpress 和 BDE/IDAPI 在笔者的机器上连接数据库的平均时间。读者可以看到，一般来说 dbExpress 在连接数据库方面比 BDE/IDAPI 迅速，dbExpress 总是比 BDE/IDAPI 更快地连接数据库。大致上 dbExpress 提供了比 BDE/IDAPI 快 30% 以上的连接效率。

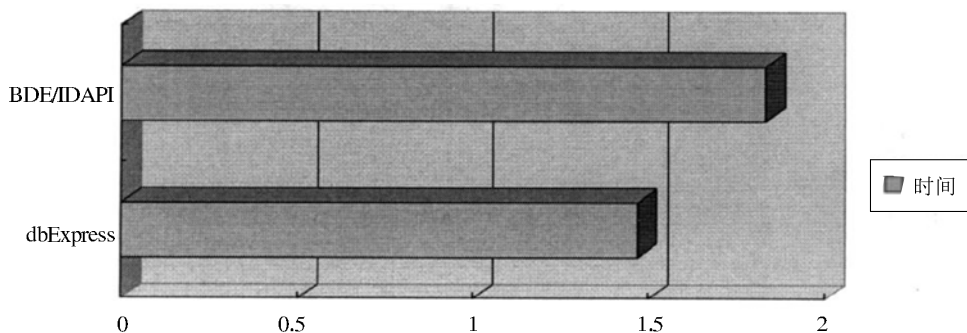


图10-11 BDE/IDAPI和dbExpress连接数据库的比较

开启数据库	dbExpress	BDE
时间	1.467	1.831

但是为什么 dbExpress 会比 BDE/IDAPI 快上 30% 呢？这是因为 dbExpress 连接数据库时只使用单向游标，而 BDE/IDAPI 会尽量使用双向游标，而且 BDE/IDAPI 会在连接数据库时先预先存储一些缓冲区和元数据，因此在连接数据库方面会比 dbExpress 缓慢一点。

10.2.2 访问大量数据的速度

在前面 10.1 小节的末尾处我们观察到了，在访问大量的数据时，BDE/IDAPI 几乎以 10 倍的效率远远超过了 dbExpress，这到底是为什么？dbExpress 在处理大量的数据时如此不堪使用吗？当然不，这是因为 dbExpress 和 BDE/IDAPI 的执行行为不一样所致。

下面的表格和图 10.12 详细比较了 dbExpress 和 BDE/IDAPI 在访问不同数量的数据时的执行结果。请注意在 BDE/IDAPI 栏中有两个数字，前面的数字是当 BDE/IDAPI 开启指定数量的记录时的结果，而后面的数字则是当 BDE/IDAPI 取得了数据之后，笔者立刻跳到最后一个记录的执行结果。

开启的记录数	dbExpress	BDE/IDAPI
10	0.097	0.061/0.101
100	0.106	0.063/0.077
1000	0.171	0.063/0.186
2000	0.233	0.064/0.261
100 000	18.052	0.064/10.135

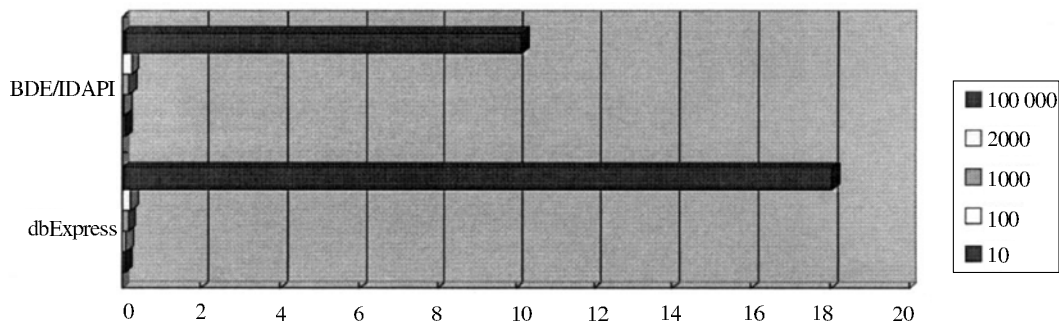


图 10-12 BDE/IDAPI 和 dbExpress 访问数据的比较

上面的表格和图 10.12 揭示了 dbExpress 和 BDE/IDAPI 的执行差异，dbExpress 之所以在前面 10.1 节末尾表现得比 BDE/IDAPI 缓慢了 10 倍以上是因为 dbExpress 在访问数据时会把所有指定的数据一次取到客户端的 DataSnap 管理的缓存内存中。而 BDE/

IDAPI则不然，BDE/IDAPI只会访问到它的内部缓冲区满了为止，之后就不再继续访问了，而当客户端需要其余尚未访问的数据时，BDE/IDAPI才会继续从数据库中取得其他数据。这就是dbExpress看起来比BDE/IDAPI缓慢了许多的原因。

因此在上面的结果表格中，笔者如果在BDE/IDAPI取得数据之后，立刻强迫BDE/IDAPI跳到最后一个记录，那么BDE/IDAPI便会从数据库中取得所有指定的数据。如果这样比较，那么我们可以看到dbExpress和BDE/IDAPI的性能就几乎是一样的了，并没有明显的差别。但是请注意，dbExpress在一次访问超大量的数据时，例如100000个记录，那么它仍然比BDE/IDAPI缓慢了许多。这是因为dbExpress在把数据传递给DataSnap时需要经过数据类型的转换，在数据量小时这还不会损失什么性能，但是数据量一大，这毕竟还是会降低性能的。因此请读者记得不要让dbExpress一次处理或是访问超量的数据。当然如果读者的应用程序需要dbExpress一次从数据库中取得100000个记录，那么笔者可以告诉读者，这大概是读者设计的应用程序有问题，读者不应该让客户端应用程序一次从数据库中取得这么多的数据。没有什么客户端引擎可以让程序员如此豪放地处理数据。

10.2.3 添加大量数据

最后再让我们看看dbExpress和BDE/IDAPI在处理修改数据时性能的差别。下面的表格和图10-13是dbExpress和BDE/IDAPI添加随机产生的数据的执行结果。

添加的记录数	dbExpress	BDE/IDAPI
10	0.052	0.036
100	0.334	0.342
1000	3.186	3.421
2000	6.514	6.732
10 000	37.992	36.109

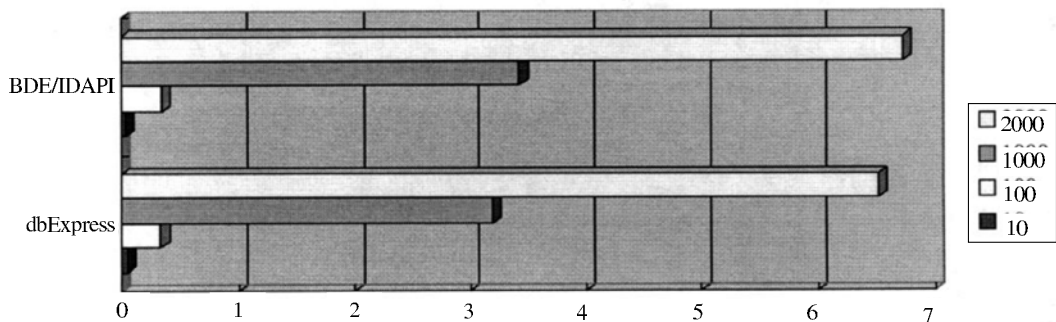


图10-13 BDE/IDAPI和dbExpress添加数据的比较

上面的执行结果可以告诉我们，dbExpress和BDE/IDAPI在处理修改的数据时没

有什么差异，几乎提供了一样的性能。但是从上面的表格中我们可以发现，当修改的数据量越来越大时，BDE/IDAPI便慢慢开始比dbExpress有效率了，这是因为如前面小节说明的dbExpress会为每一个修改的记录动态产生SQL语句。因此在默认情况下dbExpress适合处理少量和中量的数据，这种执行特性也非常符合一般的数据库应用程序和Web应用程序，不过如果读者的应用程序需要处理大量数据，那么读者一定要使用稍后章节讨论的调整dbExpress的技术。

从上面的分析和讨论中可以知道，dbExpress和BDE/IDAPI几乎提供了一样的性能。但是dbExpress可以跨平台的特性则是BDE/IDAPI无法比拟的，此外dbExpress的结构更适合在多层和Web的解决方案中使用。更何况在下面的章节中将告诉读者如何调整dbExpress，让它比BDE/IDAPI更有效率。

10.3 调整dbExpress访问数据的方式

虽然dbExpress的性能在第一个版本中就已经能够和BDE/IDAPI并驾齐驱，不过我们仍然能够从dbExpress中压榨出更多的效率，因为笔者相信没有人不希望自己编写的应用程序能够执行得更快。

dbExpress并不像BDE/IDAPI那样可以通过BDE Administrator调整许多参数。但是我们仍然能够通过简单地调整TClientDataSet组件的PacketRecords特性值，以及改变dbExpress处理数据的方式来大幅提升dbExpress的性能。

10.3.1 调整PacketRecords特性值

对于DataSnap而言，PacketRecords特性值是一个非常重要的特性。在前面的章节中本书已经说明过PacketRecords特性的意义，也告诉了各位尽量不要使用它的默认值1，以避免DataSnap把所有数据一次下载到客户端。特别是对于有大量数据的数据表而言，如此做会让你的应用程序执行得非常缓慢。

那么PacketRecords特性值到底要设置为多少才是最有效率的呢？这个问题简单的答案就是尽量只一次访问你的应用程序需要的数据，这是非常实际的答案。但是PacketRecords毕竟是有个较好的临界值，在这个临界值之内DataSnap几乎提供了一样的性能，超过这个临界值则会让性能迅速降低，因此如何找到这个临界值便是程序员的重要工作。

由于DataSnap在传递数据时会进行类型转换，因此会降低一些性能。因此程序员不应该让DataSnap一次访问超过临界值的数据，以避免额外的类型转换性能损失。下面的表格以及图10.14是使用不同的PacketRecords特性值访问测试数据表的执行结果。从这些结果中我们可以观察到一个现象，那就是当PacketRecords特性值在100之内时，DataSnap提供了几乎一样的效率，但是一超过100，性能便下降了3倍，到了

2000时效率再度下降了1倍，当然 1的设置值则得到了令人无法接受的缓慢速度。

PacketRecords (100 000个记录)	时 间
10	0.027
100	0.027
1000	0.088
2000	0.171
1	18.052

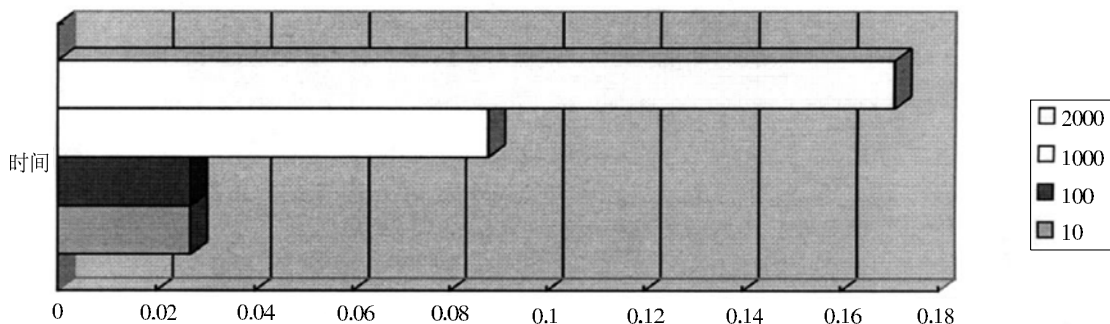


图10-14 dbExpress的PacketRecords特性对于访问数据的影响

从上面的结果中我们可以看到 100似乎是PacketRecords的临界值，超过这个值程序员就需要付出无谓的效率代价。此外，一次访问超过 100个记录也没什么意义。因此，程序员应该把PacketRecords特性值设置为100之内，这样可以使DataSnap提供最好的性能比。

当然，100是笔者的测试结果，一般的情形也是把PacketRecords设置在100之内是比较有效率的，但是如果读者的数据表的每一个记录拥有众多的字段，或是字段包含了大量的数据（例如BLOB类型的字段），那么读者应该考虑再把PacketRecords特性值减少。

10 3 2 改变dbExpress处理数据的行为

在前面比较dbExpress和BDE/IDAPI的应用中，笔者都是使用从Delphi 6开始提供的TSQLClientDataSet组件，虽然在Delphi 7中在默认情况下不会再安装TSQLClientDataSet，但是Borland为了维持Delphi 7和Delphi 6以及Kylix 1/2的兼容性，仍然允许程序员安装TSQLClientDataSet组件继续使用。为什么Borland不再继续使用TSQLClientDataSet组件呢？这是因为TSQLClientDataSet组件的性能并不好，因此在Delphi 7中才改用TSimpleDataSet来提供与TSQLClientDataSet类似的功能。

在本章稍后的章节中会继续讨论TSimpleDataSet组件，在本小节中先让我们讨论使用TClientDataSet和TDataSetProvider组件来执行添加数据的工作，用以比较

TClientDataSet加TDataSetProvider组件和TSQLClientDataSet以及BDE/IDAPI之间的异同。

图10 15显示了我们在原先的测试程序的 dbExpress数据模块中加入了 TClientDataSet和TDataSetProvider组件，并且加入了一个 TSQLQuery组件，让TSQLQuery连接到与数据模块中 TSQLClientDataSet相同的数据表，再把 TDataSetProvider连接到新加入的TSQLQuery组件，并且把TClientDataSet连接到TDataSetProvider组件。

最后，我们把添加数据要执行的 SQL Insert语句设置在 TSQLQuery的SQL特性值中：

```
insert into perftest
(ID, NAME, PHONE, ADDRESS, SALARY, EDATE
values
(:ID, :NAME, :PHONE, :ADDRESS, :SALARY, :EDATE
```

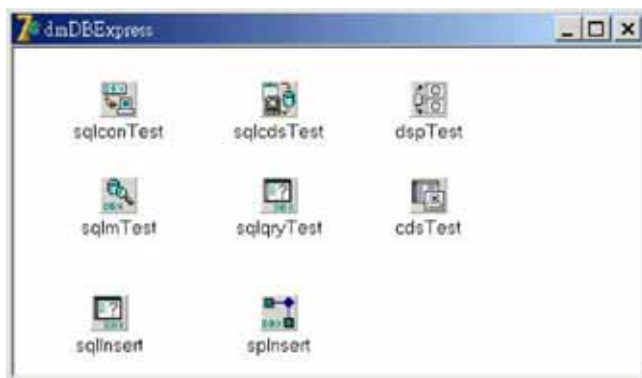


图10-15 在测试应用程序的数据模块中加入TDataSetProvider和TClientDataSet组件

1. 观察dbExpress的执行行为

现在让我们使用新加入的 TClientDataSet和TDataSetProvider组件来执行添加数据的工作，并且同样使用 TSQLMonitor来观察 TClientDataSet加TDataSetProvider组件执行时与 TSQLClientDataSet组件的执行行为有什么不同？

下面是 TSQLMonitor观察到的 TClientDataSet和TDataSetProvider组件在添加数据时执行的SQL命令：

```
INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
select * from PERFTEST

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
```

```

SELECT 0, '', '', A.RDB$RELATION_NAME, RDB$INDEX_NAME, RDB$FIELD_NAME,
B.RDB$FIELD_POSITION, A.RDB$INDEX_TYPE, EA, RDB$UNIQUE_FLAG,
C.RDB$CONSTRAINT_NAME, B$CONSTRAINT FROM RDB$INDICES
RDB$INDEX_SEGMENTS, L OUTER JOIN RDB$RELATION_CONSTRAINTS
A.RDB$RELATION_NAME C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE 'PRIMARY
KEY' WHERE (A.RDB$SYSTEM_FLAG < 0 OR A.RDB$SYSTEM_FLAG = 0) AND
(A.RDB$INDEX_NAME, RDB$INDEX_NAME) = (A.RDB$RELATION_NAME, RDB$
('PERFTEST')) ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_retaining

```

如果读者比较上面的 SQL 命令和前面小节 TSQLClientDataSet 执行的 SQL 命令，那么可以发现 TClientDataSet 加 TDataSetProvider 组件可以避免不断地重复产生 SQL 语句以及重复不必要的访问元数据动作。由于避免了这些工作，因此 TClientDataSet 加 TDataSetProvider 应该会比使用 TSQLClientDataSet 有效率得多了。下面的执行结果也证明了这个观察结果（见图 10 16）。

添加的记录数	dbExpress	BDE/IDAPI	改良的 dbExpress
10	0.052	0.036	0.047
100	0.334	0.342	0.206
1000	3.186	3.421	1.19
2000	6.514	6.732	2.686
10 000	37.992	36.109	17.472

从上面的表格数据以及图 10 16 中可以发现惊人的现象，那就是当我们使用了改良的 dbExpress 处理数据的方式之后，添加数据的性能不但比原来的方式快了许多，也比直接使用 BDE/IDAPI 更有效率，甚至当修改的数据较多时，改良的 dbExpress 的

性能比原本使用 TSQLClientDataSet 组件以及 BDE/IDAPI 快了 100% 以上。这实在很惊人, 因为虽然改良的 dbExpress 使用了类似 3 层的结构, 即使用 TClientDataSet 加上 TDataSetProvider 组件, 但是却比使用主从结构的 BDE/IDAPI 快了 1 倍以上, 这一点就可以证明 Borland 对于 dbExpress 的设计是正确的, 就是让数据库访问引擎更简单, 就可以取得更好的数据访问效率。而把缓冲数据和处理数据游标的工作交给客户端游标管理机制, 在 Delphi 中当然就是 DataSnap 了, 这样可以提供比传统数据访问引擎 (BDE/IDAPI、ODBC 等) 更好的性能。

因此读者应该了解, 如果 dbExpress 应用程序需要进行大量数据处理的工作, 那么读者应该自行使用 TSQLDataSet/TSQLQuery 等组件来执行 SQL 语句, 而避免直接使用 TSQLClientDataSet/TSimpleDataSet。因为这可以避免 TSQLClientDataSet/TSimpleDataSet 重复产生相同的 SQL 语句以及访问相同的元数据。

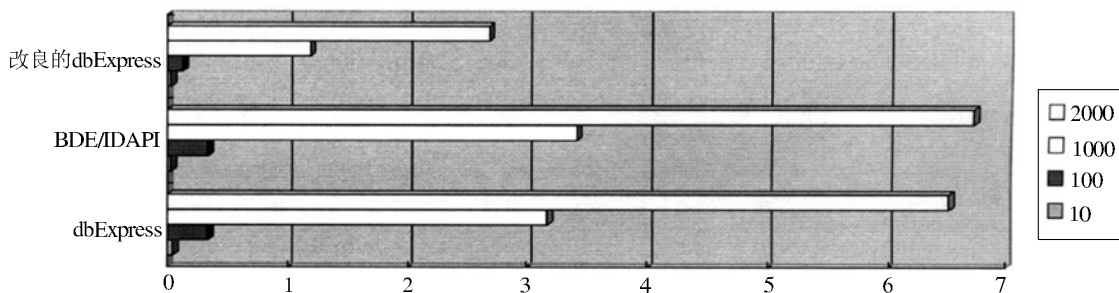


图10-16 使用改良的dbExpress处理数据的结果

现在再让我们继续比较 dbExpress、BDE/IDAPI 以及 InterBase 原始的访问引擎 IBExpress。从理论上来说, 既然是访问 InterBase, 那么原始访问引擎应该是最快的, 不过事实却不一定如此。下面的表格和图 10-17 比较了四种不同的引擎执行的结果数据。

添加的记录数	dbExpress	BDE	改良的dbExpress	IBExpress
10	0.052	0.036	0.047	0.1
100	0.334	0.342	0.206	0.709
1000	3.186	3.421	1.19	20.28
2000	6.514	6.732	2.686	69.06
10 000	37.992	36.109	17.472	1500.54

从上面的数据中我们可以看到, dbExpress 不但不比原始 IBExpress 缓慢, 而且在处理大量数据时比 IBExpress 快了数百倍。这是因为 IBExpress 在设计时是以访问少量数据为主要目标的, 不适合用来访问大量的数据。另外 dbExpress 实现得实在很有效率, 因为 dbExpress For InterBase 驱动程序也是使用 InterBase 的原始 API 来访问数据, 与 IBExpress 使用的方式是一样的, 因此 IBExpress 并没有占到任何便宜。再加上 DataSnap 处理客户端缓存内存的方法比 IBExpress 好多了, 因此 IBExpress 自然就比不

上dbExpress了。

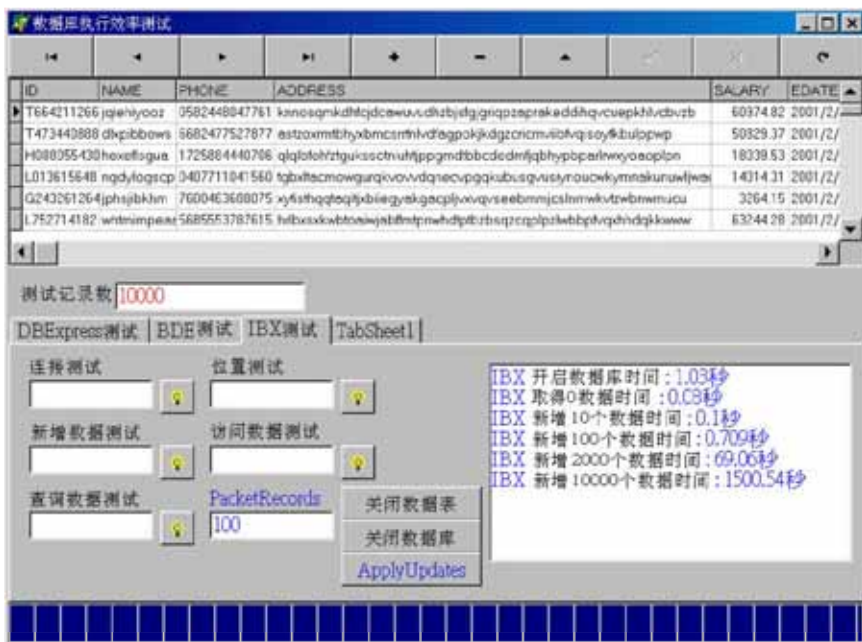


图10-17 使用原始IBExpress组件处理InterBase数据的结果

2. 选择dbExpress数据集组件

在刚才调整性能的范例中，我们使用了 `TSQLQuery`，而没有使用 `TSQLDataSet` 或是 `TSQLTable` 组件，这是为什么呢？这是因为每当 `dbExpress` 要添加一个记录时，都需要重复产生 `Insert Into` 这个SQL语句，再由数据库编译和执行，笔者希望当这个范例数据库应用程序处理修改的数据时能够尽量减少这种不必要的动作。由于这个动作会对每一个记录执行一次，因此当处理大量的数据时，这就非常浪费资源并且性能会非常缓慢。

因此要让这个范例数据库应用程序执行得更有效率，我们必须想办法让 `dbExpress` 只在一开始时为每一个要修改数据的动作产生一个相关的SQL语句，并且编译这个SQL语句以等待应用程序稍后使用它。稍后，当应用程序执行时，只需要重复执行这个已经编译过的SQL语句，如此一来不但可以提高性能，减少使用的资源，更能够大幅降低数据库服务器的负荷，让数据库服务器更有效率地为客户端服务，这样不就一举数得了吗？问题是如何事先在应用程序中准备好修改数据所需的SQL语句呢？

其实这非常简单，只需在数据模块的 `OnCreate` 事件处理函数中调用 `TSQLQuery` 的 `Prepare` 方法，以便让数据库能够预先编译客户端要执行的SQL语句，产生SQL Plan 并且在数据库中分配即将使用的资源。最后，程序员还必须记得要在数据模块的

OnDestroy事件处理函数中调用 TSQLQuery的Unprepare方法以释放在数据库中使用的资源。

```
procedure TdmdbExpress.DataModuleCreate (Sender: TObject);
begin
    ...
    sqlqryTest.Prepared := True;
    sqlInsert.Prepared := True;
end;

procedure TdmdbExpress.DataModuleDestroy (Sender: TObject);
begin
    sqlInsert.Prepared := False;
    sqlqryTest.Prepared := False;
    ...
end;
```

10.4 快速查询数据

在许多数据库应用程序中经常需要在数据表中搜寻数据，如何快速搜寻数据已经在前面的第4章“搜寻数据”中详细说明了。但是除了第4章中讨论的技巧之外，仍然有许多数据库应用程序直接使用 SQL语句来搜寻数据，或是从数据表中取得需要的信息。例如，当我们处理大量的顾客数据时，可能经常需要到订单文件中搜寻，看看某一个顾客是否在某个时间下过订单。

这种应用经常会发生在数据库应用程序中，假设现在需要处理 1000个顾客，而在处理每一个顾客时都需要到订单文件中搜寻数据，那么读者要如何解决这个问题呢？当然，读者可以使用 TSQLClientDataSet加载所有订单文件的数据，再使用 Locate来搜寻数据，但是这样做并不聪明，因为订单文件很可能拥有大量的数据。另外一种做法是当处理每一个顾客时就使用一个 TSQLClientDataSet的CommandText下达SQL命令到订单文件中搜寻数据，这样做会比使用 Locate有效率，但是还有更快速的方法吗？

当然有。TSQLClientDataSet会使用双向的游标并且允许用户修改数据，因此使用了一些额外的资源。但是对于上述的情形，到订单文件中只是查询数据，或是从订单文件中取出数据。对于这些进行大量查询的应用来说，基本上只需要使用单向、只读的游标即可，而无需使用双向游标。而 dbExpress组件使用的刚好是单向、只读的游标，因此正适合在这种应用中使用。

让我们使用下面的范例来说明使用正确的组件来进行大量的查询工作会有多大的差异。图10-18是一个使用不同dbExpress组件在数据表中进行多次数据查询的范例使

用的数据模块。在这个数据模块中使用了一个 TSQLClientDataSet组件scdsData从数据表中访问数据。然后分别使用另外一个 TSQLClientDataSet组件scdsQueryData和 TSQLQuery组件sqlqQueryData, 从一个拥有 50 000个记录的数据表中查询数据。而 scdsQueryData和sqlqQueryData都使用如下的SQL语句:

```
Select * from PERFTEST Where ID = :ID
```

在这个范例中我们要根据随机产生的 ID (身份证ID) 值到PERFTEST数据表中取得NAME (姓名) 字段的值。由于只是要取得 NAME字段值并且显示出来, 因此也可以使用TSQLQuery组件来查询。



图10-18 范例查询应用程序的数据模块

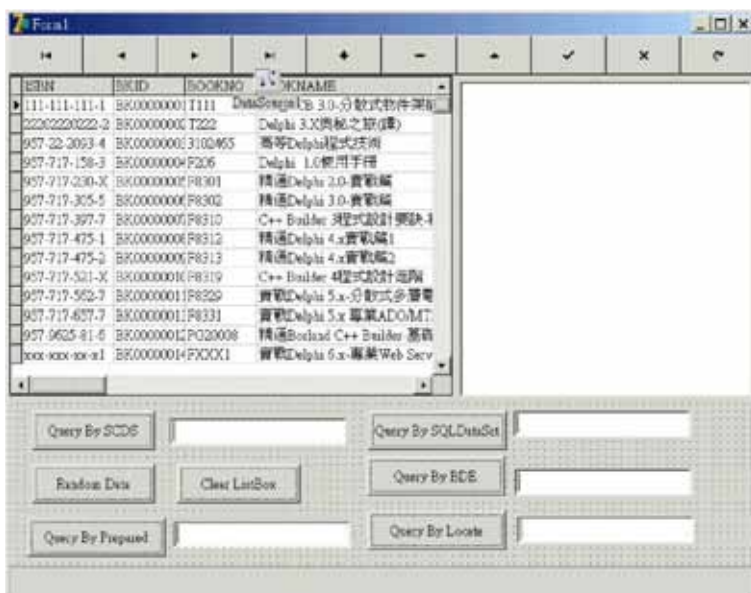


图10-19 范例查询应用程序的主窗体

在范例应用程序的主窗体中有四个按钮，其中 Random Data按钮会动态产生一组从PERFTEST数据表中随机获取的ID数据（见图10-19）。而其他三个按钮则分别使用TSQLClientDataSet组件、TSQLQuery组件以及BDE/IDAPI的TQuery组件使用这组随机的ID数据再从PERFTEST数据表中取得NAME字段的值，并且填入到主窗体中的TListBox中。其中“Query By SCDS”按钮使用TSQLClientDataSet组件，“Query By SQLDataSet”按钮使用TSQLQuery组件，“Query By BDE”按钮使用BDE/IDAPI的TQuery组件。

以下的程序代码分别是“Query By SCDS”按钮以及“Query By SQLDataSet”按钮的OnClick事件处理函数，从程序代码中可以看到这两个事件处理函数除了使用的dbExpress组件不一样之外，所有其他程序代码都是一样的。

```

procedure TForm1.btnscdsClick (Sender: TObjekt;
var
    iCount : Integer;
    iTime : Integer;
begin
    lbData.Items.Clear;
    SetStartTime;
    for iCount := 1 to ILOOP do
        begin
            dmQueryData.scdsQueryData.Active := False;
            dmQueryData.scdsQueryData.Params.ParamByName('ID').Value := slData.Strings
[iCount - 1];
            dmQueryData.scdsQueryData.Active := True;
            lbData.Items.Add(dmQueryData.scdsQueryData.FieldByName('NAME').Value);
        end;
        SetEndTime;
        iTime := GetRunTime;
        edtQueryByscds.Text := FloatToStr(iTime / 1000.00) + '秒';
    end;

procedure TForm1.btnsqlqClick (Sender: TObjekt;
var
    iCount : Integer;
    iTime : Integer;
begin
    lbData.Items.Clear;
    SetStartTime;

    for iCount := 1 to ILOOP do
        begin
            dmQueryData.sqlqQueryData.Active := False;

```

```
dmQueryData.sqlqQueryData.ParamByName('ID').Value := slData.Strings[iCount - 1];
dmQueryData.sqlqQueryData.Active := True;
lbData.Items.Add(dmQueryData.sqlqQueryData.FieldByName('NAME').Value);
end;
SetEndTime;
iTime := GetRunTime;
edtQueryBysqlq.Text := FloatToStrTime / 1000.00 + '秒';
end;
```

为什么这个范例要使用 **TSQLQuery**？因为 **TSQLQuery** 是唯一开放 **Prepared** 特性值的 **dbExpress** 组件，通过 **Prepared** 特性值，程序员可以预先编译要频繁执行的 SQL 语句，而取得较好的性能。

Borland 在 Delphi 6 最后的版本（RTM, Release To Manufacturing）中也开放了 **TSQLDataSet** 的 **Prepared** 特性，因此读者也可以使用 **TSQLDataSet** 了，当然在 Delphi 7 中是完全没有问题的。

因此在这个范例的数据模块中也使用了下面的程序代码，让 **TSQLQuery** 组件预先编译它使用的 SQL 语句：

```
procedure TdmQueryData.DataModuleCreate(Sender: TObject);
begin
    sqlqQueryData.Prepared := True;
end;

procedure TdmQueryData.DataModuleDestroy(Sender: TObject);
begin
    sqlqQueryData.Prepared := False;
end;
```

图 10 20 是范例应用程序执行的结果，我们可以看到使用 **TSQLClientDataSet** 查询需要花上 6.354 秒，而 **TSQLQuery** 只需要 1.203 秒，快了将近 6 倍。而 **BDE/IDAPI** 则需要 4.826 秒。

图 10 21 是开启 **BDE/IDAPI** 的预先编译功能后再次查询的结果，我们看到 **BDE** 快了 1 秒多的时间，也就是改善了将近 35% 的效率。

图 10 22 是把 **TSQLQuery** 的预先编译功能关闭后再次查询的结果，我们看到 **TSQLQuery** 的执行时间从 1.215 秒增加到 1.273 秒，仍然比 **BDE** 全速的速度快了将近 3 倍。这个结果可以充分证明在查询数据时单向、只读游标的速度确实很快。

最后让我们看看，如果直接使用 **Locate** 方法执行相同的查询，而且预先加载 **PERFTEST** 的所有数据，再进行查询的结果。从图 10 23 中可以看到，即使是让 **dbExpress** 组件预先加载所有数据，再使用 **Locate** 查询，其执行速度仍然比上述的方

法缓慢了许多。

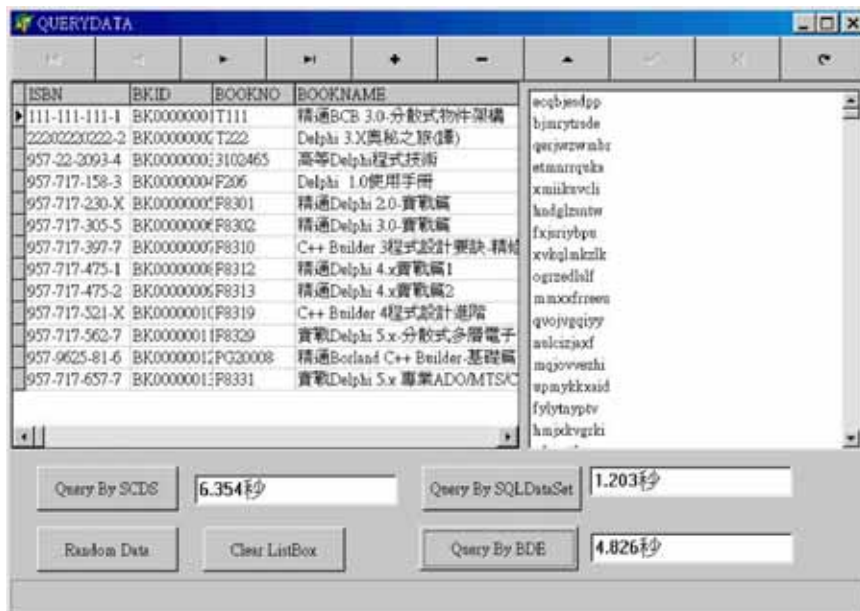


图10-20 范例查询应用程序的执行结果

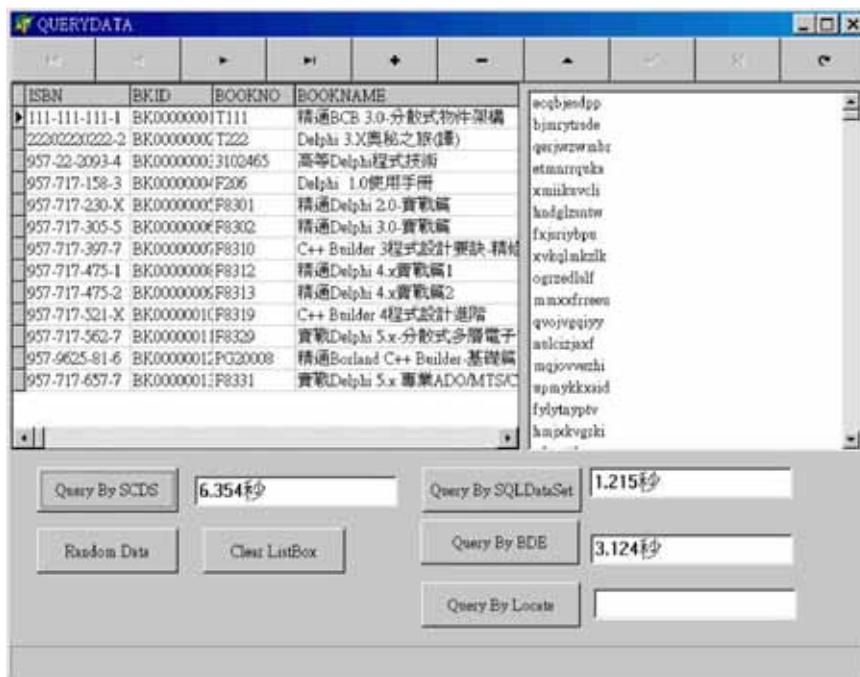


图10-21 让BDE也预先编译要执行的SQL语句

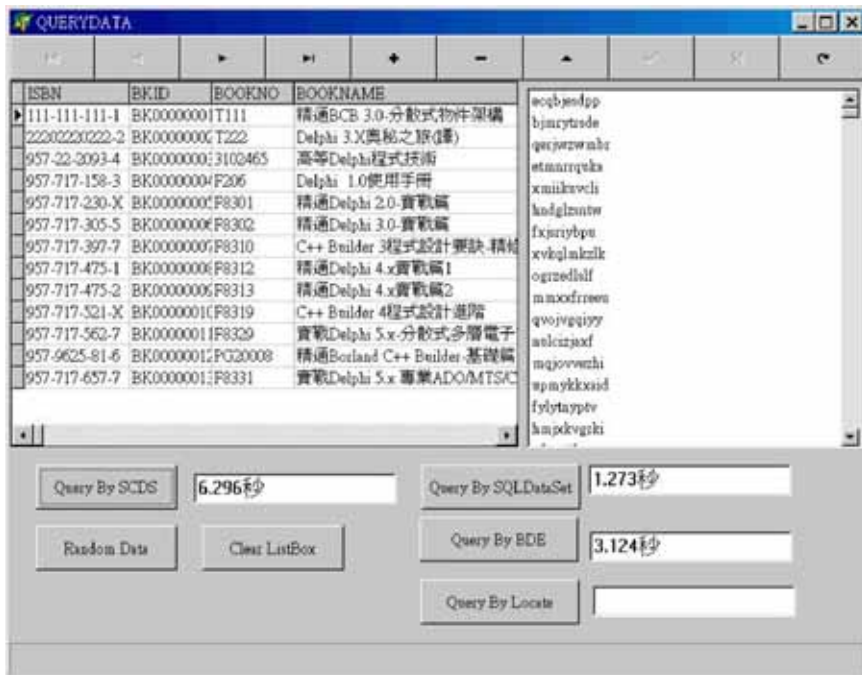


图10-22 关闭TSQLQuery的预先编译功能

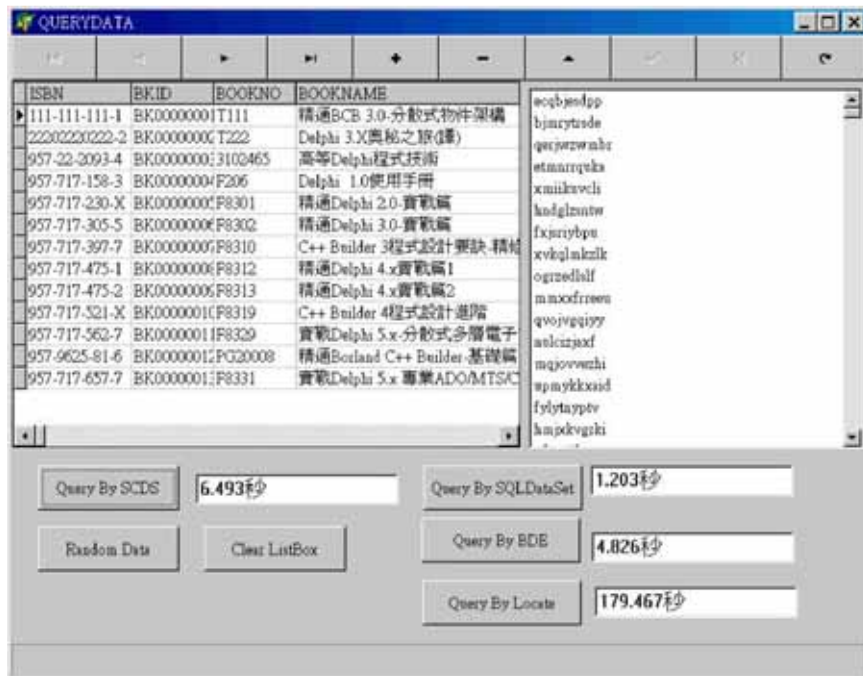


图10-23 直接使用 Locate方法进行相同的查询工作

在上面的范例中，我们直接使用 TSQLQuery 组件来进行数据查询的工作，而且这些查询到的数据只能读取而无法修改。这是因为 dbExpress 组件使用单向、只读的游标。如果应用程序不但需要查询大量的数据，而且也希望能够修改这些查询到的数据，那么是不是也能够获得如此大幅度的性能改善呢？

当然可以，只要我们结合使用 TSQLQuery 和 TDataSetProvider 组件就可以取得接近直接使用 TSQLQuery 查询数据的速度。例如，图 10-24 是在范例应用程序中再建立一个数据模块，并且在数据模块中直接使用 TSQLQuery 和 TDataSetProvider 组件来进行数据查询的工作。其中的 sqlqPrepared 也使用下面的 SQL 语句：

```
Select * from PERFTEST Where ID =
```

在相同的查询条件下，我们可以看看避免使用 Delphi 提供的 TSQLClientDataSet 组件来进行大量数据查询的执行结果。

图 10-25 是使用 TSQLQuery 搭配 TDataSetProvider 组件执行相同查询的结果。从图中可以看到，自行巧妙地搭配 TSQLQuery 和 TDataSetProvider 组件仍然能够取得很好的性能，只比直接使用 TSQLQuery 组件缓慢了一点点，但是好处是程序员可以使用一致的程序开发模型，而且查询到的数据仍然可以进行修改。



图10-24 使用TSQLQuery搭配
TDataSetProvider组件

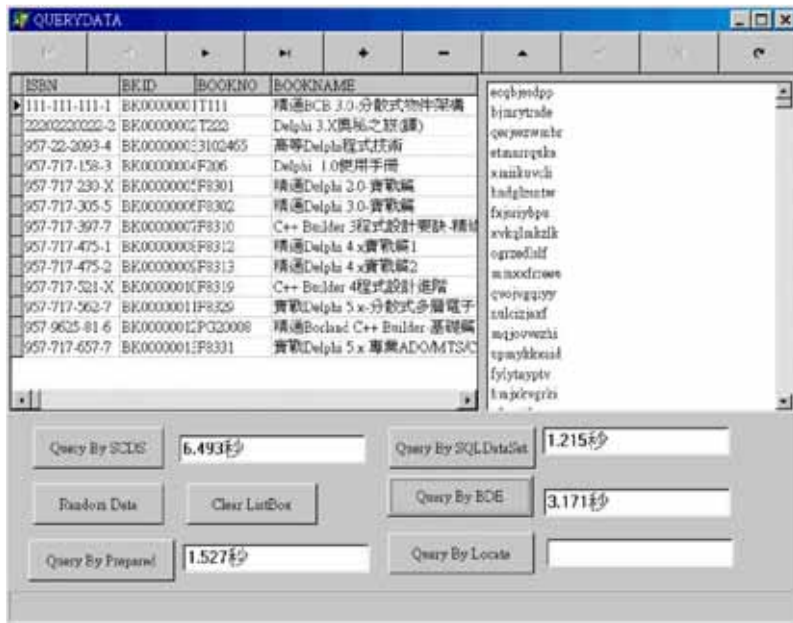


图10-25 使用TSQLQuery组件搭配TDataSetProvider组件进行查询的结果

本小节说明了当应用程序需要进行大量数据查询的工作时，如何使用比较有效率的方式来查询数据。正确地使用 dbExpress 组件将为你的应用程序带来数倍的性能。

10.5 Delphi 7 之后的 TSimpleDataSet

Borland 为了改善 Delphi 6/Kylix 中 TSQLClientDataSet 性能不佳的问题，在 Delphi 7 中建议程序员除非是为了兼容性的原因，否则就不应该再使用 TSQLClientDataSet，而使用 TSimpleDataSet 来代替。TSimpleDataSet 避免了 TSQLClientDataSet 中许多无效率的地方，并且提供了接近 TClientDataSet 加 TDataSetProvider 的性能，因此可以说在性能和使用的方便性上都提供了很好的效果。不过就如同前面章节在许多地方讨论的一样，程序员如果想要取得最大的弹性以及最好的性能，那么使用 TClientDataSet 加 TDataSetProvider 才能拥有最好的效果。

10.6 结论

本章对于 Delphi/Kylix 的 dbExpress 组件进行了深入的讨论，除了说明 dbExpress 组件和引擎的执行特性和重要的概念之外，也详细说明了 dbExpress 的执行行为。为什么要了解 dbExpress 的执行行为呢？不了解这些 dbExpress 技巧会有什么妨碍吗？当然，如果程序员不知道这些详细的技术，那么仍然可以使用 dbExpress 开发数据库应用程序，而且在执行上也不会有什么太大的问题。但是在一些处理大量数据的应用上则会显得力不从心，性能缓慢。了解 dbExpress 的特性除了可以让程序员开发出更有效率的应用程序之外，也能够有一些应用上通过这些知识克服应用系统的难点，或是克服 dbExpress 的局限性而轻松地完成应用系统。

dbExpress 的执行行为是程序员必须了解的重要技巧。除此之外，如何选择使用适当的 dbExpress 组件在应用程序中也是一个关键问题。Delphi/Kylix 提供了一整套 dbExpress 组件让程序员使用。对于一般的应用来说，几乎都需要对数据进行查询和修改的处理，不过大部分 dbExpress 组件都是无法修改数据的组件，只有 TSQLClientDataSet 可以同时进行所有种类的数据处理工作，因此我相信大部分程序员都只使用 TSQLClientDataSet 组件，可能不去使用其他 dbExpress 组件，也不知道为什么要使用其他 dbExpress 组件。事实上，每一个 dbExpress 组件都有其特定的功能，正确地使用 dbExpress 组件可以大幅度增加应用程序的性能，并且减少应用程序需要使用的资源，因此本章也讨论了如何使用 dbExpress 组件来进行适当的数据处理工作。

在阅读完本章之后，程序员应该对于 dbExpress 有了深入的了解，并且知道如何快速地使用 dbExpress 开发出最有效率的应用程序了。

第11章 动动脑，快乐一下

前面的章节讨论了许多 dbExpress 相关技术，读者现在应该对于 dbExpress 有了一定的了解和掌握。在第 10 章中也讨论了许多有关 dbExpress 性能的主题，如果读者是从第 1 章开始阅读本书，就会发现在许多章节中已经陆陆续续讨论了许多增加 dbExpress 性能的内容，并不仅限于第 10 章讨论的内容。在本章中本书将使用两个简单的场景提出问题，让读者也能够动动脑筋，看看如何使用本书讨论的概念和技巧来尝试解决本章的仿真问题。

11.1 从一个看似简单的场景开始

笔者日前突然接到一位读者寄来的一封信。在信中这位读者有如下的一段话：

李先生您好，我一直是您的读者，非常喜欢您的书，也希望您能够一直为 Delphi 程序员编写更多的好书，谢谢。是这样的，目前我正在开发一个数据库应用系统，在这个系统中我遇到了一个问题，那就是我的系统必须在一定的时间之内从仪器中接收大量的数据并且添加到数据库中。由于仪器产生的数据量非常大，因此我的程序必须能够正确地把数据添加到数据库中，而不能遗失数据，我想请问的是有什么方法可以满足这种需求？ Delphi 6/7 的 dbExpress 能够应付这个状况吗？例如，如果我想在 5 分钟内处理 1 000 000 个记录，那么我该如何做？

第 2 个问题是当数据进入数据库之后，如果我想从其中获取特定的数据，那么我应该如何做才能够以最有效率的方式取得这些数据？

希望您能够给我一些建议，再次谢谢您。

在看到这封信时我也没有什么想法，因为 dbExpress 虽然比 BDE 有效率，但是是否能够在一定的时间之内承受大量的数据也是一个未知数。另外，是否只使用 dbExpress 就能够处理这样大量的数据笔者也没有把握（不过我想一定是不行的，不然这位读者直接用 dbExpress 就可以了，也不需要写信给笔者了）。如果直接使用 dbExpress 不行的话，那么又有什么方法可以解决呢？

这些问题不但引起了笔者的兴趣，笔者也想趁机测试一下 dbExpress 在处理大量数据时的表现。现在就让我们开始和这位读者一起来尝试解决问题，并且运用我们掌握的 dbExpress 知识吧。

11.2 开始动动脑吧

dbExpress到底能够以多快的速度处理数据呢？在前面的章节中，我们已经可以确定dbExpress比BDE速度快，而且Borland会继续增加dbExpress的功能和效率。但是dbExpress到底能够多快地把数据添加到数据库中？是否能够跟上这位读者的仪器发出数据的速度？现在就让我们直接试试dbExpress的速度吧。为了模拟仪器在处理数据时花费的时间，我们建立一个仿真的数据表，并且在程序中以计算的方式来产生每一个字段的值，故意增加程序执行时间以模拟实际情况，最后再计算整体的执行结果。

1. 第一个尝试，试试dbExpress的极限

第一步当然是使用dbExpress来试试dbExpress的执行速度，下面的程序代码直接使用TClientDataSet和dbExpress在InterBase中添加数据，而且每一个字段值都是经过计算产生的：

```
procedure TfrmPerfMain.btndbExpressClick(Sender: TObject);
var
    iCount: Integer;
begin
    ILOOPS := StrToInt(edtTestCount.Text);
    dmDBExpress.cdsTest.DisableControls;
    pgLoops.Position := 0;
    pgLoops.Max := ILOOPS;

    try
        LogStartTime;
        for iCount := 1 to ILOOPS do // Iterate
            begin
                dmDBExpress.cdsTest.Insert;
                dmDBExpress.cdsTest.FieldByName('ID').Value := GetID;
                dmDBExpress.cdsTest.FieldByName('NAME').Value := GetName;
                dmDBExpress.cdsTest.FieldByName('PHONE').Value := GetPhone;
                dmDBExpress.cdsTest.FieldByName('ADDRESS').Value := GetAddress;
                dmDBExpress.cdsTest.FieldByName('SALARY').Value := GetSalary;
                dmDBExpress.cdsTest.FieldByName('EDATE').Value := Now;
                dmDBExpress.cdsTest.Post;
                pgLoops.Position := pgLoops.Position + 1;
                Application.ProcessMessages;
            end; // for
        dmDBExpress.cdsTest.ApplyUpdates(0);

        LogEndTime;
```

```
LogRunTime (mmStatus, 'DB新增' + IntToStr(ILOOPS) + '笔数据时间 : ');  
finally  
    dmDBExpress.cdsTest.EnableControls;  
end;  
end;
```

执行上面的程序之后，我们可以得到图 11-1 所示的结果。从画面显示的数据可以计算出直接使用 dbExpress 来处理数据时 1 秒只能处理大约 295 个记录。

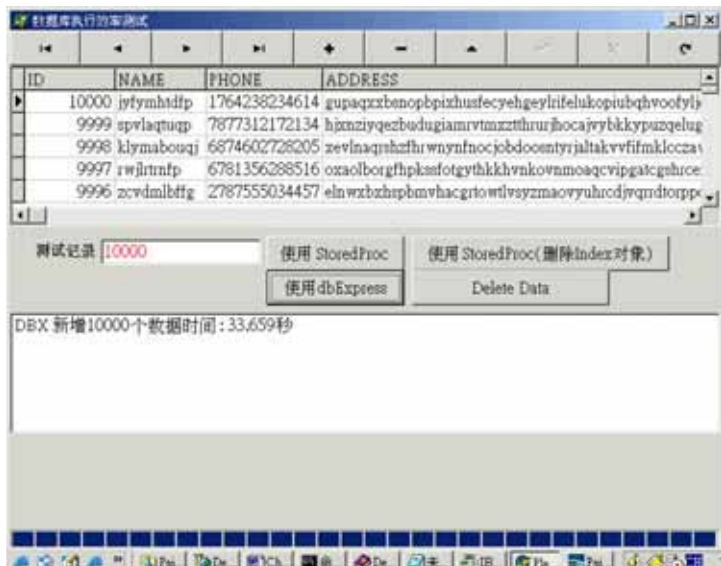


图11-1 使用dbExpress处理大量数据

以这种速度来处理 1 000 000 个记录的话，总共需要 57 分钟左右，这和 5 分钟的要求实在是差太多了。而且在上面的程序中，所有数据是先暂时存储在 TClientDataSet 的内存中，最后只使用了一个数据库事务把数据添加进数据库中，造成的负荷已经很小了。看来尽管 dbExpress 已经比 BDE 等速度更快，但是直接使用 dbExpress 仍然无法满足这位读者的要求。那么我们如何改善这个执行速度呢？

2. 第二个尝试，使用存储过程

当然，我们可以修改上面的程序，让它直接使用 TSQLDataSet 组件用 SQL 语句直接添加数据，这样做的执行速度相信又会快上许多。不过使用 TSQLDataSet 组件仍然比不上使用存储过程，因为我们可以利用数据库的存储过程特性让执行的速度更为迅速。

因此我们尝试的第二个方法就是使用存储过程来应付庞大的数据量。在前面的章节中，本书已经讨论过如何通过 dbExpress 中的 TstoredProc 组件来调用存储过程，因此现在我们就直接使用下面的程序代码来处理相同的数据，看看使用存储过程之后

的执行结果:

```

procedure TfrmPerfMain.btnSPClick (Sender: TObject;
var
    iCount: Integer;
    TD: TTransactionDesc;
begin
    ILOOPS := StrToInt(edtTestCount.Text);
    dmDBExpress.cdsTest.DisableControls;
    pgLoops.Position := 0;
    pgLoops.Max := ILOOPS;

    try
        if not dmDBExpress.sqlconTest.InTransaction then
            begin
                TD.TransactionID := 1;
                TD.IsolationLevel := xilREADCOMMITTED;
                dmDBExpress.sqlconTest.StartTransaction (TD);
                try
                    LogStartTime;
                    for iCount := 1 to ILOOPS do      // Iterate
                        begin
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('ID').Value := GetID;
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('NAME').Value := GetName;
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('PHONE').Value :=
                                GetPhone;
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('ADDRESS').Value :=
                                GetAddress;
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('SALARY').Value :=
                                GetSalary;
                            dmDBExpress.spAddLargeDatas.Params.ParamByName ('EDATE').Value := Now;
                            dmDBExpress.spAddLargeDatas.ExecProc;
                            pgLoops.Position := pgLoops.Position + 1;
                            Application.ProcessMessages;
                        end;      // for
                    dmDBExpress.sqlconTest.Commit (TD); {on success, commit the changes};
                    LogEndTime;
                    LogRunTime (mmStatus, 'StoredProc 新增' + IntToStr (ILOOPS) + '笔数据时间 : ');
                except
                    dmDBExpress.sqlconTest.Rollback (TD); {on failure, undo the changes};
                end;
            end;
        finally
            dmDBExpress.cdsTest.EnableControls;
        end;
    end;

```

使用相同的数据产生方式，但是通过存储过程把数据添加到数据库中果然大幅度增加了处理的数据量。在这次测试中添加 10 000个记录只花了 6.739秒，比刚才快了将近5倍，可以说进步是很大的（见图 11 2）。

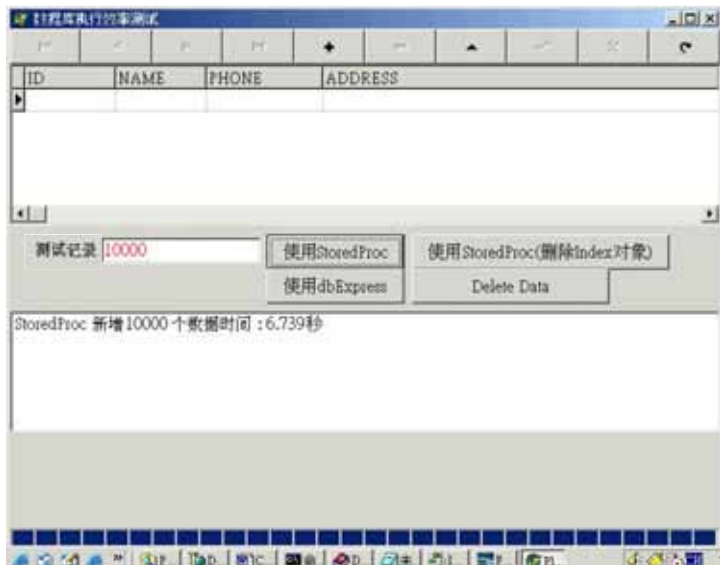


图11-2 使用dbExpress调用存储过程处理大量数据

图 11 3显示了直接使用 dbExpress 和使用存储过程处理 10 000 个记录的时间差异。

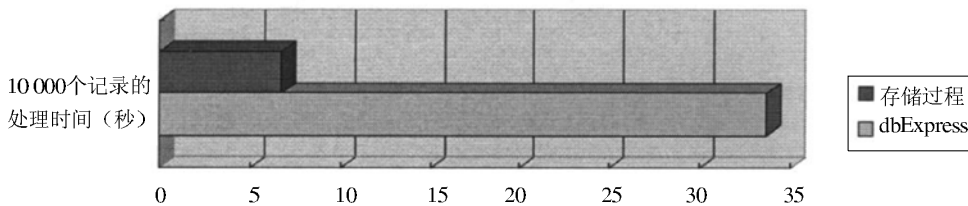


图11-3 使用dbExpress和使用存储过程处理数据的比较

如果我们计算这两种方式在 1 秒之内可处理的数据量，会发现直接使用 dbExpress 时 1 秒只能处理 200 多个记录，而使用存储过程时却能处理超过 1 400 个记录。单位时间内可处理的数据量实在是差异颇大（见图 11 4）。

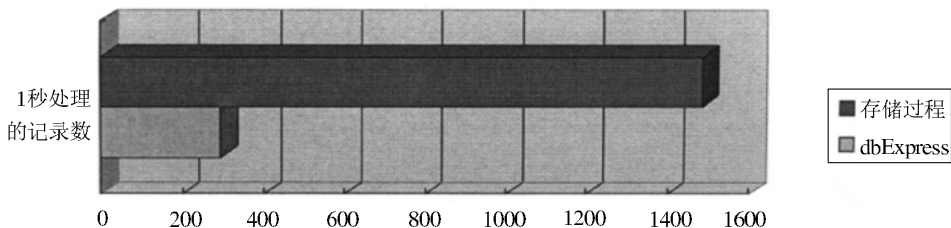


图11-4 使用dbExpress和使用存储过程在1秒内可处理的数据量比较

虽然使用存储过程大幅度增加了处理的速度，但即使是以存储过程在 1 秒内可处理的数据量来计算，我们似乎仍然离这位读者希望的数据处理量差了一大截，虽然存储过程几乎是我们已知最快速的数据处理方式。那么我们有没有更快的方法可以继续接近我们的目标？

3. 还不够好，能有其他方法吗？

想想当我们在前面把数据添加到数据库中时，除了数据本身之外，还有哪些东西也会被写入数据库中？如果我们能够尽量减少其他额外的写入数据，那么不是就可以增加性能了吗？没错，这个想法就是本小节想要达成的目的。

在应用程序把数据写入数据库的同时，一般来说在数据库中也会为数据建立索引等信息，如果添加数据的数据表定义了许多索引，那么添加数据的时间也会增加。而当数据是以随机的方式产生时，除了需要花费建立索引的时间之外，数据库中存储索引的节点也会不断地增加和分裂，更进一步增加了需要的处理时间。随着添加的数据增多，数据库也会需要更长的时间来处理。

了解了这个原理之后就让我们想想，在这种需要大量随机添加数据的应用中，数据库也一定会花费大量的时间为数据建立和维护索引，因此如果我们能够尽量减少（甚至避免）在添加数据的过程中数据库不断重复处理索引的时间，那么添加数据的总时间不就可以节省许多吗？有了这个想法之后，就让我们开始试验这个概念。

首先在我们添加大量数据之前，让我们先删除这个数据表的索引信息，让数据库只负责把数据添加到数据表中，这样可以避免数据库把时间花费在不断维护索引上。等到应用程序把所有数据添加完毕之后再让数据库一次性建立索引，这样可以让数据库以最有效率的方式建立索引信息。因此在下面的程序代码中当应用程序开始添加数据之前，它调用 `pDeleteIndexes` 函数先把数据表的索引删除，等到数据添加完毕之后再调用 `pCreateIndexes` 重新为所有数据建立索引信息。请注意，我们一定要把这两个函数封装在 `try...finally...end` 语句中以保证最后索引一定会被重新建立。

```
procedure TfrmPerfMain.btnspIndexClick(Sender: TObject);
var
    TD: TTransactionDesc;
    iCount : Integer;

    procedure pDeleteIndexes;
    begin
        if not dmDBExpress.sqlconTest.InTransaction then
        begin
            TD.TransactionID := 2;
            TD.IsolationLevel := xilREADCOMMITTED;
            dmDBExpress.sqlconTest.StartTransaction(TD);
            dmDBExpress.sqldsGeneral.CommandText := DELETEINDEX;
            dmDBExpress.sqldsGeneral.ExecSQL(True);
```



```

        dmDBExpress.sqlconTest.Commit (TD) ; {on success, commit the changes};
    end;
end;

procedure pCreateIndexies;
begin
    if not dmDBExpress.sqlconTest.InTransact then
    begin
        TD.TransactionID := 3;
        TD.IsolationLevel := xilREADCOMMITTED;
        dmDBExpress.sqlconTest.StartTransaction (TD) ;
        dmDBExpress.sqldsGeneral.CommandText := ADDINDEX;
        dmDBExpress.sqldsGeneral.ExecSQL (False) ;
        dmDBExpress.sqlconTest.Commit (TD) ; {on success, commit the changes};
    end;
end;
begin
    ILOOPS := StrToInt (edtTestCount.Text) ;
    dmDBExpress.cdsTest.DisableControls;
    pgLoops.Position := 0;
    pgLoops.Max := ILOOPS;

    pDeleteIndexies;
    try
        if not dmDBExpress.sqlconTest.InTransact then
        begin
            TD.TransactionID := 1;
            TD.IsolationLevel := xilREADCOMMITTED;
            dmDBExpress.sqlconTest.StartTransaction (TD) ;
            try
                LogStartTime;
                for iCount := 1 to ILOOPS do      // Iterate
                begin
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('ID') .Value := GetID;
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('NAME') .Value := GetName;
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('PHONE') .Value :=
                        GetPhone;
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('ADDRESS') .Value :=
                        GetAddress;
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('SALARY') .Value :=
                        GetSalary;
                    dmDBExpress.spAddLargeDatas.Params.ParamByName ('EDATE') .Value := Now;
                    dmDBExpress.spAddLargeDatas.ExecProc;
                end
            end
        end
    end
end

```

```
pgLoops.Position := pgLoops.Position + 1;  
Application.ProcessMessages;  
end;    // for  
dmDBExpress.sqlconTest.Commit(TD); {on success, commit the changes};  
except  
    dmDBExpress.sqlconTest.Rollback(TD); {on failure, undo the changes};  
end;  
end;  
finally  
    pCreateIndexies;  
    LogEndTime;  
    LogRunTime(mmStatus, 'StoredProc(Index) 新增' + IntToStr(PLoops) + ' 笔数据时间 : ' );  
    dmDBExpress.cdsTest.EnableControls;  
end;  
end;
```

现在使用新的方法添加 10 000个记录，看看这个方法是否真的能够增加性能。图 11 5是执行这个方法之后的结果，我们可以看出这个方法果然比刚才的存储过程更有效率。

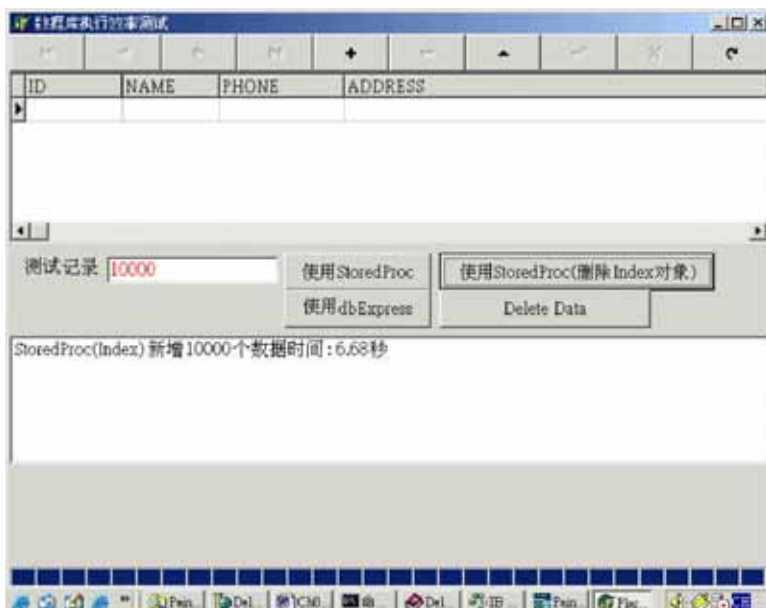


图11-5 使用新的方法可继续增加处理数据的速度

现在我们把到目前为止使用的三个方法一起比较，会发现我们的确是在不断地进步（见图 11 6）。

现在第三个方法在单位时间之内仍然增加了可处理的数据量（见图 11 7）。

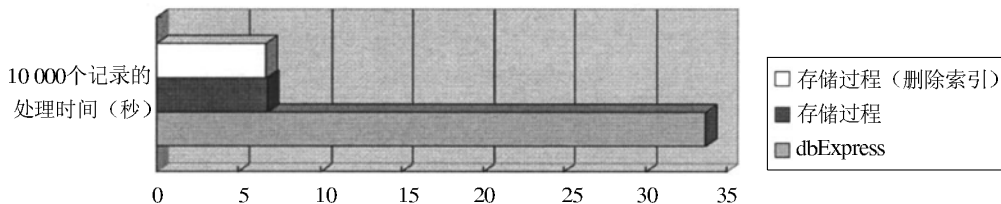


图11-6 新的方法可以使用更少的时间处理 10000 个记录

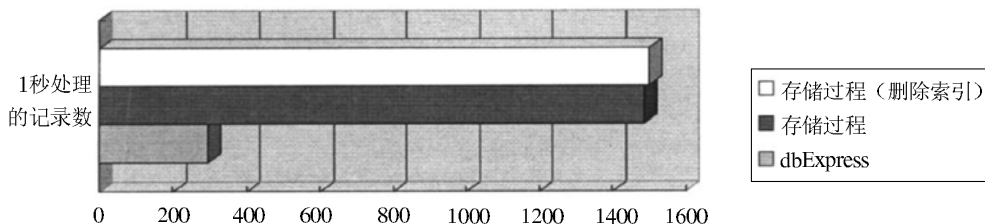


图11-7 新的方法在单位时间内可处理更多的数据

虽然本小节使用的删除索引方法的确可以继续增加处理数据的速度，但是似乎可以增加的性能已经极为有限了，那么到底使用这个新的方法划不划算呢？如果读者有这个想法的话，那么就代表读者已经开始一起参与这个有趣的试验了。基本上在这个范例中，10 000个记录还不算多，因此在不大的数据量中先删除索引，添加完数据之后再建立索引的方式能够增加的性能是有限的，甚至有可能造成处理数据更为缓慢的现象。但是如果在大量数据的情形中，比如前面这位读者来信说的1 000 000个记录，那么会如何呢？这就值得我们研究了。

4. 大量数据测试的结果

当添加的数据更多时，第二种处理数据的方式就需要花费更多的时间来处理数据和索引。因此现在让我们使用更多的数据量来测试，看看使用第三种方式处理数据是不是值得。

图11 8和图11 9分别显示了使用存储过程和结合存储过程以及删除索引信息来处理200 000个记录的结果。

从图11 10中我们可以看到，当数据量到达200 000个记录时，结合存储过程和删除索引的处理方式非常明显优于只使用存储过程。我们增加了将近一倍的效率。

图11 11显示的信息更为有趣。图11 11中显示了使用存储过程处理200 000个记录时，1秒内只能处理将近800个记录。现在请读者回头参考图11 7，就可以发现随着数据量越来越多，数据库可在单位时间之内处理的数据量就随之快速减少。这当然是因为数据库花了更多的时间处理数据本身之外的工作，而删除索引的方式避免了数据库不断地花时间维护索引，因此即使在处理200 000个记录的情形下仍然能够提供接近第二种方法处理10 000个记录时的速度，算是相当优秀的。基本上读者也可以使用这个范例来测试读者使用的数据库在处理大量数据时处理速度衰退的幅度。

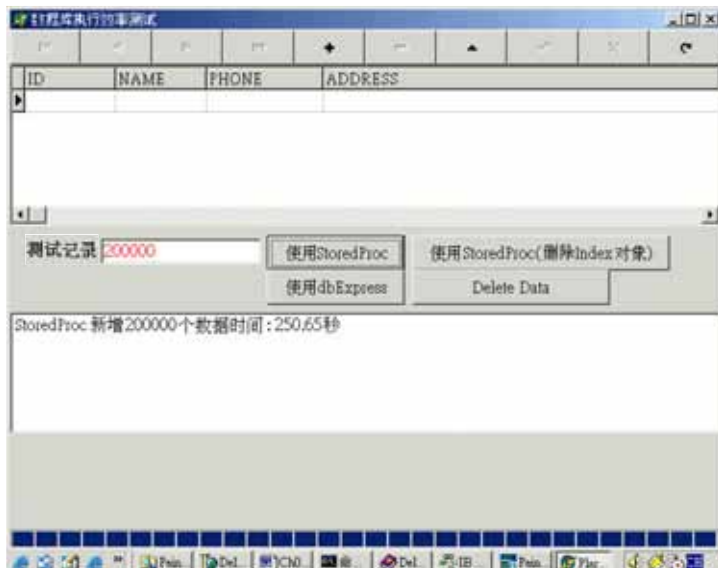


图11-8 使用存储过程处理200 000个记录需要花费的时间

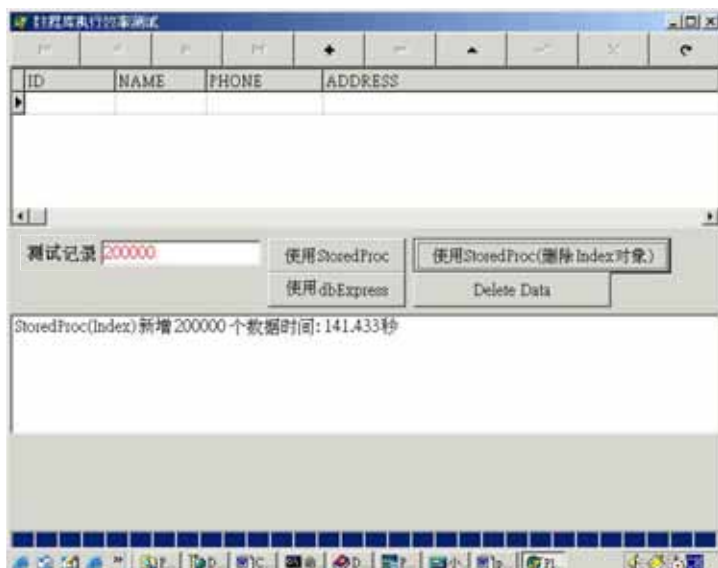


图11-9 使用存储过程结合删除索引处理200 000个记录需要花费的时间

从本小节的测试中我们可以知道，使用结合存储过程和删除索引方法在处理大量数据时可以大幅增加单位时间内处理的数据量。在处理 200 000个记录时，第三种方法比第二种方法快了几乎一倍。基本上，数据量越大，数据表拥有的索引信息越多，第三种方法就越明显地优于只使用存储过程，因此相当适合在需要大量处理数据的应用中使用。

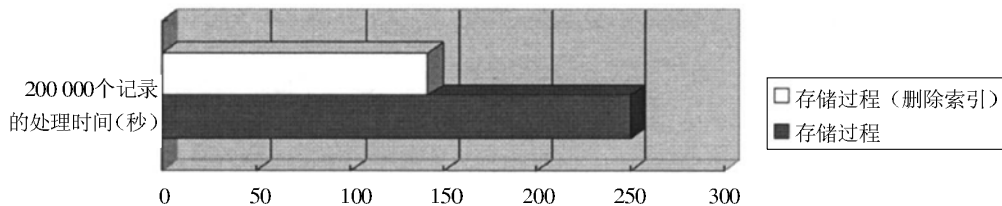


图11-10 在大量数据处理时，第三个方法明显优于只使用存储过程

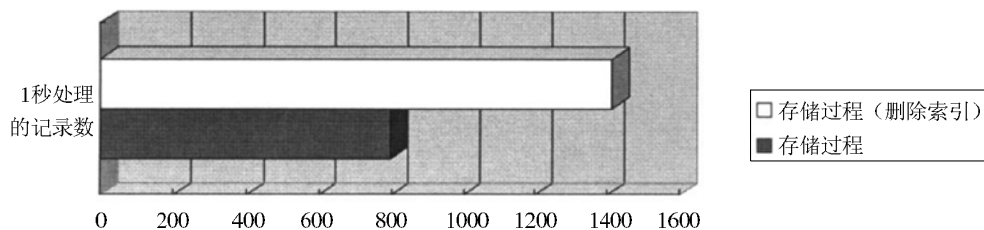


图11-11 第三个方法在单位时间内大幅增加了可处理的数据量

5. 我们失败了吗？

从图11-11中可以看出，我们只花了142秒左右就处理了200 000个记录，换算一下在1秒之内几乎可处理1 400多个记录，算是相当的理想，而且笔者的机器不算高档，如果使用服务器级的机器，那么肯定可以有更好的表现。但是除了前面讨论的方法之外，有没有更快的方式能够让数据进入数据库中？

许多人在初次从纯文件数据库转换到关系型数据库时，总是喜欢以添加大量数据时的性能来质疑为什么使用关系型数据库比纯文件数据库更缓慢。这当然是因为关系型数据库在修改数据时除了把数据添加到数据库之外，还会进行许多其他工作，例如维护索引、数据一致性以及数据安全等。而纯文件数据库大都只是把数据添加到数据库而已，进行的其他工作比较少，自然也就比较快了。但是纯文件数据库有其局限性，例如在这个范例中要处理数百万个记录，那么肯定到最后纯文件数据库会比不上关系型数据库，不管是在处理数据的速度、数据安全和稳定性上都会如此。

在这个范例中，如果我们想要进一步增加处理数据的速度，那么我们可以进一步使用特定的关系型数据库提供的功能，例如 Bulk Transfer、Block Copy、二进制大量数据移动和拷贝等更具效率的转移方式，把数据从客户端添加到数据库服务器中。当然在使用这些更有效率的数据移动方式时，也一定要配合暂时停止索引工作的机制，如此一来应该能够再次提高数据处理的速度。

此外，适当增加数据库服务器使用的内存、增加数据库服务器的数据缓存内存大小、调整网络性能等也有助于提高数据处理的速度。

最后，这位提出问题的读者也应该计算一下使用的硬件能够提供的最大数据处理速度，例如内存的速度、硬盘驱动器的速度以及网络瞬间提供的最大速度。也许在

所有软件手段运用完之后，剩下的就是硬件的限制了，当然也就不属于本书讨论的范围了。

11.3 第二个问题

这位读者的第二个问题非常类似于本书在前面章节中讨论的如何有效率地处理和搜寻数据的主题。这位读者想要知道当大量数据进入了数据库之后，如何能够有效率地在这些随机数据中选取他需要的数据？

例如，这位读者想要在 1 000 000 个记录中找出所有薪资在 10 000 到 11 000 之间的信息，那么我们应该如何处理？从前面章节的讨论中我们知道似乎可以在程序中建立一个新的 `TClientDataSet` 组件，克隆原本存储数据的 `TClientDataSet` 组件的游标，再使用新的 `TClientDataSet` 组件来过滤用户想要的信息，这样做比直接使用原本存储数据的 `TClientDataSet` 组件快。因此我们可以使用如下的程序代码来取得数据：

```
procedure TfrmPerfMain.Button2Click(Sender: TObject);
var
  aTCDS : TClientDataSet;
begin
  LogStartTime;
  aTCDS := TClientDataSet.Create(Self);
  try
    aTCDS.CloneCursor(dmDBExpress.cdsTemp, False, False);
    aTCDS.Filtered := False;
    aTCDS.Filter := 'Salary >= ' + Self.ledtLowerSalary.Text + ' and Salary <= ' + Self.ledtUpperSalary.Text;
    aTCDS.Filtered := True;
    Self.ledtFilterCount.Text := IntToStr(dmDBExpress.cdsTemp.RecordCount);
    Self.ledtNFilterCount.Text := IntToStr(aTCDS.RecordCount);
  finally
    aTCDS.Free;
  end;
  LogEndTime;
  LogRunTime(mmFilter, 'Dynamic Filter 耗时 : ');
end;
```

不过读者应该知道，在这种大量数据的情形下，我们最好还是最优先考虑如何有效率地访问数据，而把图形用户界面的问题放在其次。因此在这个情形下笔者的建议还是直接使用另外一个新的 `TClientDataSet`，直接以用户的条件从后端数据源中取得用户需要的数据，而不要管已经存储数据的 `TClientDataSet` 组件，这样才会最有效率。而且在这种情形下应用程序也一定要检查用户设置的条件不可太宽松，如果条

件很宽松一定要提醒用户，并且使用回调（Call Back）机制允许用户在觉得执行时间太久时中断应用程序的执行。最后再配合 TClientDataSet 的 PacketRecords 一次只访问少量的数据，那么整体性能应该就会很合理。因此下面的程序代码应该会比较理想的做法：

```
procedure TfrmPerfMain.Button3Click(Sender: TObject);
begin
    LogStartTime;
    dmDBExpress.cdsGeneral.Active := False;
    dmDBExpress.cdsGeneral.CommandText := 'select * from LARGEDATAS where Salary >= ' + Self.leDtLowerSalary.Text + ' and Salary <= ' + Self.leDtUpperSalary.Text;
    dmDBExpress.cdsGeneral.Active := True;
    Self.leDtFilterCount.Text := IntToStr(dmDBExpress.cdsTemp.RecordCount);
    Self.leDtNFilterCount.Text := IntToStr(dmDBExpress.cdsGeneral.RecordCount);
    dmDBExpress.cdsGeneral.Active := False;
    LogEndTime;
    LogRunTime(mmFilter, 'Dynamic Filter耗时 : ');
end;
```

当我们使用这两个不同的方法来查询数据时，可以很明显地看出它们的差异。例如，图 11-12 和图 11-13 显示了第一个方法需要花费 70 多秒，同时会在客户端的内存中存储 200 000 个记录；而使用第二个方法只需要花费 1.6 秒多，差了将近 40 倍的性能。如果读者是这个系统的用户的话，那么读者会希望使用哪一个应用程序？答案已经相当明显了。

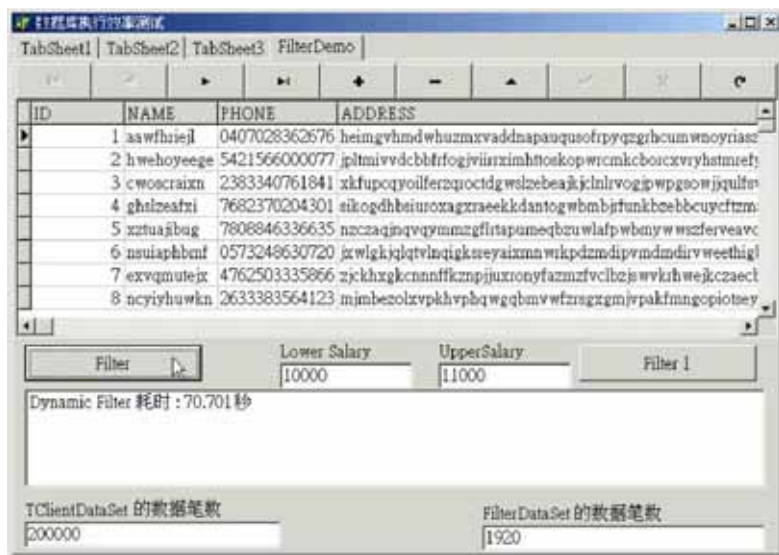


图11-12 以SQL语句来选择所有数据

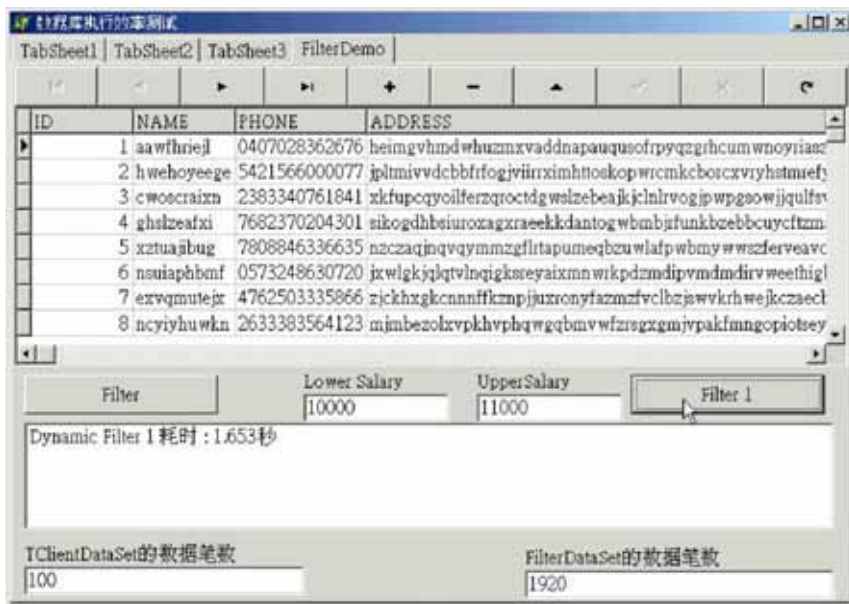


图11-13 重新使用TClientDataSet直接从后端数据源中选择数据

从读者的这两个问题中我们可以知道，即使在使用相当有效率的 dbExpress时，程序员仍然需要花一点儿脑筋想想如何最有效率地处理数据，而不能只是光靠 dbExpress。

例如，如果读者是在 WAN或是缓慢的网络环境中使用 dbExpress，那么除了应该避免无谓的数据访问和网络的往返之外，还要记得避免访问无谓的元数据、避免建立无谓的数据库连接等。遵循这些良好的数据访问习惯可以让读者的数据库应用系统跑得又快又稳。

11.4 结论

本章的讨论方式也算是别出心裁，和许多中文书籍不太一样吧。本章从一个看似简单的问题开始试着解决某位读者想要解决的问题。在初步尝试之后，我们很快发现事情并没有想象的那么简单，于是我们开始尝试寻找是否有其他技术能够帮助我们解决问题。在尝试数个方法之后，我们找到了一些比原本快数倍的数据处理方法，相当令人印象深刻。而且从这个解决问题的过程中我们也了解到光靠软件本身是不够的，一定要结合经验和技巧才有可能解决原本不可能完成的任务。在最后，本章也讨论了一些或许能够再增加性能的技术，不过这些技术已经和特定的数据库有关了，读者可以依据自己使用的数据库来进一步想办法向目标挑战。

本章的第二个问题和前面章节讨论的如何有效率地处理和搜寻、排序数据有关。读者从讨论的内容中可以知道，只需要根据我们对于 dbExpress 技术的了解就可以找到既可以增加性能又可以减少成本的方法。

本章的内容是以比较轻松的方式让读者能够通过某位读者提出的问题而试着使用本书讨论的概念和技术寻找解答。笔者希望本章讨论的主题能够触发读者更多有趣的想法或概念来解决在系统设计方面的问题。

第12章 数据访问技术

本书至此已经算是接近尾声了，前面的章节已经详细介绍了如何使用 dbExpress 开发数据库应用程序。dbExpress是一个新的数据访问技术，也将是 Borland未来努力的重点技术之一。虽然dbExpress还很年轻，但是未来dbExpress将会快速并且持续地改善。

不过使用 Delphi的程序员可能使用各种不同的数据库，有的使用关系型数据库，有的使用基于文件的数据库，也有各自的应用。dbExpress基本上是一个以处理 SQL命令的数据访问引擎，它和BDE源自Paradox访问引擎可以说是采用完全不同的设计方式（因为BDE是从Paradox访问引擎引进来的，而dbExpress采用了和BDE完全不同的设计结构），因此dbExpress可以说是以能够处理SQL命令的关系型数据库为主，因此dbExpress也有适用的范围。

本章的内容主要是讨论程序员应该选用哪一种技术来访问各种不同的数据源，选择使用适当的数据访问引擎和技术对于应用系统来说是非常重要的，因为这将会决定应用系统是否能够有效率地处理数据，甚至会影响应用系统的稳定性。下面的章节是根据作者的经验总结出来的，读者可以作为参考，以作为选择数据引擎的根据之一，读者在实际选用数据访问引擎和技术时仍然应该有自己的选用标准。

12.1 dbExpress的发展

目前在 Kylix 3和Delphi 7中的dbExpress是2.0版，然而Delphi 7的dbExpress已经修正了Kylix 1/Delphi 6中dbExpress的许多问题并且增加了许多新的dbExpress驱动程序和功能，因此严格地说 Delphi 7/Kylix 3的dbExpress应该是2.x的版本。

现在 Borland正在持续地强化dbExpress的功能，Borland开发完成了dbExpress For MS SQL Server 2000，使用的技术是直接封装OLE DB让dbExpress通过OLE DB来访问MS SQL Server。这是一个非常好的解决方案，因为ADO和OLE DB几乎是现在访问MS SQL Server 2000最好且唯一提供完整支持能力的连接技术。dbExpress For MS SQL Server直接封装OLE DB，所以不但可以使用MS SQL Server的所有功能，更能够提供比ADO更具效率的处理能力，也可以在MTS和COM+中执行，利用MTS/COM+的数据库缓冲池功能。但是如此一来dbExpress For MS SQL Server将只能在Windows平台中执行，因此在Kylix 3中并没有提供MS SQL Server的dbExpress驱动程序。

此外, dbExpress For Sybase也已经在dbExpress的开发计划中, Borland准备开发dbExpress For Sybase system 12以提供连接Sybase的驱动程序。

Delphi 7/Kylix 3的dbExpress可连接MySQL 3.23.49版, 但是MySQL 4之后的版本在API方面有了改变, 造成了dbExpress无法正确地访问MySQL 4.x, 现在Borland正在开发新版本的MySQL驱动程序, 相信应该会很快地推出, 供用户下载。

12.2 BDE的状况

现在BDE已经进入维护的状态了, 除了修改bug之外Borland几乎不会再对BDE进行任何改善。Delphi 7中的BDE应该是最后一个改善的版本, 当dbExpress成熟而且具备完整的数据库驱动程序之后, BDE将会走入历史。

12.3 ADO

ADO是Microsoft主推的数据访问引擎, MS SQL Server和Access也可以使用ADO来访问, 许多数据库厂商也推出了ADO驱动程序, 例如Oracle和Sybase等。ADO自从2.5版之后的2.6/2.7版也只是修改bug, 已经没有什么重大的改善。不过ADO是Windows平台上最重要的数据访问技术之一已经是毋庸置疑的了。但是在.NET推出之后, 由于应用程序的开发方向逐渐走向XML/SOAP和Web Service的类型, 因此在.NET中ADO的数据也逐渐以XML的形式封装, 并且以ADO.NET作为标准的访问技术。因此笔者估计日后ADO将以更好的XML支持为持续改善的方向, 并且逐渐淡出Microsoft的主流数据访问技术。

12.4 可选用的数据库

各种数据库和不同的数据访问技术组合后的数目的确庞大, 如何做出适当的选择的确是令人头痛的问题。数据访问技术不断变化除了是因为信息技术的进步之外, 新的应用也是主因之一。不光是Borland在改变数据访问技术, Microsoft更是不断地改变数据访问技术。从特定的数据库API, 到ODBC, OLE DB到ADO, 一直到.NET中的ADO.NET, 这些变化不会停止, 因为信息技术原本就在不断地改善。现在在Windows平台上只有Microsoft和Borland有能力提出数据访问技术的标准, 因此在Windows平台上也就是以这两个厂商的解决方案为选择的标准。而在Linux\UNIX平台上SUN是主导的力量。在Linux平台上, 因为SUN怕Linux和Solaris竞争, 所以SUN原本不是很积极, 因此当Borland在Linux上推出了dbExpress技术之后, 反而提供了更好的数据库访问方式。因此在Linux上如果不使用Java, 那么dbExpress将会拥

有重大的影响力。所以可供选择的主要数据访问技术的范围大概就是 Microsoft、SUN和Borland的标准，再加上一些其他第三方特定的解决方案。重要的是，读者必须了解如何搭配数据库和适当的数据访问技术。下面的表格列出了重要的数据库以及适当的数据访问技术供读者参考。

数 据 库	数据访问引擎	说 明
InterBase	dbExpress、IBO或IBX	InterBase是非常开放的数据库，几乎拥有最多的数据访问技术，但是目前IBO和dbExpress仍然是最好的
MS SQL Server	dbExpress、ADO、OLE DB、ASP.NET	除了ADO/OLE DB和dbExpress之外，目前没有其他更适合的数据访问技术
Informix	dbExpress、BDE	Informix已经在市场中消失了，dbExpress和BDE应该是剩下来最好的解决方案了
Sybase	dbExpress、ADO、BDE	逐渐失去市场影响力，现在 ADO和BDE是比较好的数据访问技术，Borland也计划加入dbExpress For Sybase System 12的驱动程序
Oracle	dbExpress、ADO、BDE、ODAC	数据库市场的老大，ADO和BDE都提供了良好的解决方案，但是未来 dbExpress将会提供最好的解决方案。另外一个访问Oracle的良好解决方案则是第三方厂商提供的ODAC组件组了
IBM DB2	dbExpress、ADO、BDE	也是相当开放的数据库，目前也是以 ADO/BDE为主要的访问方式，不过dbExpress将逐渐提供更好的解决方案
MySQL	dbExpress	对Delphi来说，除了第三方组件之外，dbExpress应该是最好的连接技术了。建议不要使用 ODBC，因为BDE的ODBC Socket 仍然有一些问题

1. InterBase

InterBase是笔者很喜欢的数据库之一，也是笔者最常使用的数据库之一。自从 Borland把InterBase的源代码开放之后，InterBase便分成三个主要的源流。一个是 Borland仍然出售的正式版 InterBase，在笔者编写本书时的最新版本是 6.5.x.x。第二个版本是开放源代码的 InterBase，这个版本可以自由使用，只要遵照 Borland的版权规范即可。第三个版本是由一个开放社团根据 Borland开放的源代码自行开发的版本：FireBird。FireBird是一个免费的 InterBase数据库，加入了一些新的功能并且修正了 InterBase开放源代码版本中的许多错误。现在 FireBird已经将进入正式的 1.x版本，这个社团未来计划继续改善 FireBird。

InterBase 6.x和FireBird都是相当好的数据库，不但简单、好用、有效率，而且可以在数个操作系统之上执行。InterBase的5.5版本有一些 bug，因此会造成数据库的损坏，不过在 InterBase 5.6之后 Borland已经修正了问题。而笔者使用 IBO配合 InterBase从来没有发生过任何问题，因此笔者个人相当喜欢 InterBase和IBO的组合，快速、稳定又方便。至于 IBX呢，笔者在先前的版本中进行了一些测试，对于 IBX的性能并不满意而且有许多小 bug在不断地修正，因此虽然 IBX是免费的，但是笔者仍

然喜欢使用 IBO 或是 dbExpress。目前 Borland 已经在研发 InterBase 7，预计在 2002 年底推出。

相关的 URL

- WWW.BORLAND.COM
- WWW.IBObjects.COM
- WWW.FirdBird.COM
- WWW.IBPhoenix.COM
- WWW.IBExpert.COM
- WWW.EMS HITECH.COM
- WWW.GIMBAL.COM

2. MS SQL Server

Microsoft 的 SQL Server 目前无疑是 Windows 上最受欢迎的关系型数据库，Oracle 在 Windows 平台上已经被 SQL Server 追上了，以往的 Sybase 和 Informix 也慢慢退出了 Windows 平台。不过 MS SQL Server 现在逐渐走向大型关系型数据库的方向，对于许多仍然适用于小型系统的应用是否适合是需要程序员考虑的。对于 MS SQL Server 来说，从一开始以 DB Library 为原始 API，到以 ODBC 为原始 API，再到以 ADO/OLE DB 为原始 API，现在逐渐以 ADO.NET 为原始 API 可以说是一直在改变的。

目前 BDE 是以调用 DB Library API 为访问 MS SQL Server 的方式，由于 Microsoft 已经不再维护 DB Library，因此许多新的 MS SQL Server 特点已经无法通过 BDE 使用了。又由于 Microsoft 没有把 MS SQL Server 的 TDS 数据格式授权给 Borland（因为 Microsoft 一直不肯公开 TDS 的功能规格），因此 Borland 无法在第一个 dbExpress 版本中编写访问 MS SQL Server 的驱动程序，但是 Borland 已经决定直接以 dbExpress 的接口封装 OLE DB 来访问 MS SQL Server，如此一来不但可以使用 MS SQL Server 的所有特点，也能够使用最有效率的方式访问 MS SQL Server，将比使用 ADO 拥有更好的性能。

.NET 中的 ADO.NET 在底层仍然是以 ADO 来访问 MS SQL Server，因此在可以预见的未来 ADO 和 OLE DB 将是访问 MS SQL Server 的原始 API，dbExpress 使用 OLE DB 来访问 MS SQL Server 是很正确的选择。现在 Borland 已经推出了 dbExpress For MS SQL Server 的驱动程序，并且也已经开始研发 dbExpress For .NET 的驱动程序，准备在未来 Borland 在 .NET 上的开发工具 Galileo 中使用。

3. Informix

再见了 Informix，这个曾经叱咤一时的数据库产品终究抵不过市场严酷的竞争而被 IBM 收购，正式走入历史。Informix 在前几年由于策略的错误导致市场不断缩小，而自行开发的数据库开发工具 NEON 又成不了气候，因此逐渐成为市场上最封闭的数据库之一，这可以从市场上的开发工具不易连接 Informix 数据库看出。虽然

Informix在后期努力改善这个现象，无奈在市场上已经时不我予。如果读者正巧是使用Informix，那么BDE似乎是仅剩的最好的连接技术了。在被IBM收购之后，Informix数据库将会逐渐消失，因为IBM一定会主推其DB2数据库，并且IBM的开发工具也没有受到市场的充分欢迎和接受，因此即使IBM的开发工具推出连接Informix的解决方案，也可能是属于IBM的特有技术，或是使用老旧的ODBC技术。因此使用最新的dbExpress For Informix驱动程序或是BDE将会是比较好的选择。

4. Sybase

自从Sybase帮助Microsoft的SQL Server推出之后就已经注定了Sybase数据库将逐渐从Windows平台上淡出，因为在Unix和Linux上Sybase竞争不过Oracle，在Windows平台上现在又比不上Microsoft的SQL Server，因此Sybase处于一个尴尬的地位，再加上Sybase的Power Builder也慢慢地失去市场，因此造成了Sybase数据库也持续失去以往的影响力。不过Sybase仍然还占有一定的市场，也推出了ADO连接技术，因此对于Sybase数据库而言，ADO或BDE都是不错的连接技术。此外Borland也计划在未来在dbExpress中加入连接Sybase System 12的驱动程序，这对于使用Sybase的软件开发人员来说的确是一个好消息。

5. Oracle

不用说Oracle目前仍然是数据库的霸主，虽然Oracle在UNIX/Linux上受到IBM强烈的挑战，在Windows平台上又逐渐被Microsoft赶上。由于Oracle是目前市场上最重要的数据库之一，因此有各种不同的连接技术。目前ADO和BDE都是连接Oracle不错的选择，但是听说Oracle将不再支持ADO，而BDE又进入维护状态，不知道是否会再提供新版Oracle的连接。不过dbExpress非常支持Oracle，Borland也将持续提供连接Oracle的解决方案，因此使用dbExpress或是ODAC连接Oracle是不错的选择。

6. IBM DB2

现在最积极的数据库厂商应该就算是IBM了，最近IBM不断促销DB2，并且在日前买下了Informix，瞬间在数据库市场占有了大量的用户人数。以往DB2基本上是在大型主机上执行的数据库，到Windows平台上之后其图形用户界面管理工具也不完善，因此并没有获得很大的市场。不过近来DB2不断改善，不但在性能上突飞猛进，在管理工具方面也有了明显的改善。目前BDE、ADO和dbExpress都能够访问DB2，算是非常开放的数据库，和Informix比起来这方面IBM是比较好的。不过对于DB2来说，虽然Delphi 7中的BDE对于DB2有许多改善，但是未来仍然是以dbExpress为主。在Delphi 7推出之后，IBM已经在DB2的开发者版本中捆绑了Borland Delphi 7 Enterprise的试用版，这意味着IBM承认在Windows平台上目前Delphi是支持DB2数据库最好的开发工具。

12.5 几种数据库及数据访问技术

许多用户仍然在使用基于文件的数据库，例如 Paradox、dBase或是其他数据库。对于这些基于文件的数据库来说，BDE是很适合的数据访问技术而且目前 Borland也没有计划提供基于文件的 dbExpress驱动程序（笔者也认为不需要，因为 dbExpress的结构设计是以访问关系型数据库为主，并不适合用来访问基于文件的数据库），当然如果读者需要完整的解决方案也可以使用第三方的数据访问技术。下面的表格总结了笔者使用过的一些数据库以及适当的数据访问技术。

数 据 库	数据访问引擎	说 明
Paradox	BDE	对于 Paradox，BDE应该是最适合，也是最原始的数据访问技术。因为 BDE就是从当初的 Paradox Engine演化而来的
FoxPro	第三方	BDE虽然支持访问 FoxPro数据库，但是仍然不完善，读者可以选择其他厂商提供的解决方案
dBase	BDE	对于 dBase来说，BDE应该是一个不错的数据访问技术，没有什么大问题
DBISAM	DBISAM	DBISAM是一个非常稳定的基于文件的数据库，它使用自己的格式，但是非常有效率而且支持 SQL，非常适合用来开发单机，主从结构的应用系统。但是不适合用来开发 N层的应用系统，因为它与 MIDAS/DataSnap有一点儿不兼容。目前 DBISAM也推出了 Client/Server版本，专门提供需要 Client/Server解决方案的数据库应用系统

12.6 数据库和组件模型

许多数据库应用系统现在已经和各种组件模型结合在一起，以提供更高的延展性以及和 Web解决方案结合。目前在 Windows和其他平台上最流行的组件模型包括 Microsoft的MTS/COM+、CORBA以及EJB，下面的小节分别简单地介绍在这些组件模型中比较适合使用的数据访问技术。

1. MTS/COM+

对Microsoft的MTS和COM+来说，除了ADO和OLE DB之外几乎没有其他更适合使用的数据库访问技术了。因为目前只有 ADO和OLE DB才能够配合MTS/COM+的数据库缓冲池技术以及分布式事务管理功能。即使是 ODBC也无法在MTS/COM+中充分地发挥功能，因为有许多 ODBC驱动程序并不能正确地在MTS/COM+中执行。

即使是ADO目前也无法完全发挥 COM+的对象缓冲池功能，因为 ADO仍然是使用Apartment线程模型，未来Microsoft将会继续改善ADO。不过总归一句话，要使用MTS/COM+组件模型，就请使用ADO或是OLE DB。

2. CORBA

对于CORBA来说也许是比较尴尬的，因为 CORBA本身虽然结构良好，但是太复

杂，一般程序员不易使用，也不易良好地运用。由于 CORBA 本身没有数据库缓冲池的机制，因此除非 CORBA 的厂商自行开发了此功能，否则站在开发者的立场最好是使用更简单、更有效率的数据访问技术。例如在 dbExpress 没有推出之前，ODBC 是不错的选择，但是现在有了 dbExpress，由于 dbExpress 可以跨平台，既简单又有效率，因此是搭配 CORBA 很好的选择。笔者看到有用户搭配使用 CORBA 和 ADO，笔者建议千万不要这么做，因为 ADO 是使用 COM 组件模型，与 CORBA 的组件模型在线程模型和实例模型方面有差异，又有不同的数据类型要转换，不但没有效率，又危险。还是使用 ODBC 或是 dbExpress 才比较适合。

3. EJB

EJB 是现在除了 COM+ 之外在 Linux/UNIX 平台上最流行的组件模型了，不过由于 EJB 几乎是纯 Java 的解决方案，因此在使用 EJB 后就几乎只能使用 JDBC 来访问数据了，没有其他选择方案。目前 JDBC 仍然还在进化中，SUN 也在为 Java 和 EJB 开发其他数据访问技术，例如数据对象（Data Object JDO）。总归一句话，就和 COM+ 一样，要使用 EJB 就请使用 JDBC。

12.7 结论

选择数据库是需要根据用户开发的应用系统特性来决定的，而且在许多时候可能是由其他人决定而不是由软件人员决定，但是使用什么数据访问技术来访问特定的数据库则是软件开发人员可以决定的，而且必须好好考虑。使用适当的数据访问技术并且了解未来技术的发展趋势是非常重要的，因为这个决定不但会影响应用系统的稳定度以及性能，也会影响到未来应用系统是否能够持续地运作下去。

希望在读者阅读完本书之后不但学习到了如何使用 Delphi 和 dbExpress 开发有效率的数据库应用系统，更能够在阅读完本章之后了解各个数据库的异同以及适当的访问技术。dbExpress 虽然是一个新的、优秀的数据访问技术，但是要不要使用它仍然必须由读者根据使用的数据库以及其他因素来决定。笔者虽然写了这本以 dbExpress 为主题的技术书籍，而且笔者相信未来 dbExpress 将会是 Borland 主要的数据访问技术，但是读者仍然需要自己做一个聪明的决定。不过不管读者的选择是什么，都希望读者能够顺利地使用 Delphi 成功地开发出理想的数据库应用系统。当然，本章的内容是根据笔者的经验总结出来的，不一定百分之百正确，因为笔者所知也有限。如果读者有更好的经验或是见解，欢迎读者告知笔者，作为日后修改本书内容的参考，谢谢。

第13章 dbExpress的实现和未来的发展

本书的内容主要是讨论如何使用 dbExpress 开发高效率的数据库应用程序，许多 dbExpress 主题已经在前面的章节中说明得非常清楚了。虽然现在读者根据前面章节讨论的内容应该已经可以很好地掌握 dbExpress 技术，但是如果能够再深入了解一点 dbExpress 的内部以及 dbExpress 在未来如何在 .NET 平台上使用也是很有帮助的。因此本章将讨论比较高级的 dbExpress 技术内容，以便让读者对于 dbExpress 如何运作有一个更清楚的了解。如果读者对于这些高级技术没有兴趣，或是觉得这些内容太过困难的话，当然也可以跳过本章讨论的内容。不阅读本章的内容并不会影响读者使用 dbExpress 的功能。

13.1 dbExpress的实现技术

Borland 决定开发新一代的 dbExpress 技术是有许多原因的，这些在前面的章节中都已经讨论过。dbExpress 在 Delphi 6/Kylix 1.0 中开始正式出现在市场上接受考验，虽然 dbExpress 是相当新的数据库访问技术，但是却发展得相当迅速，这是因为 Borland 内部有一个专门的小组在研发 dbExpress 相关的技术。

对于程序员来说，除了了解如何有效率地使用 dbExpress 之外，事实上看看 Borland 如何定义 dbExpress 标准和开发 dbExpress 技术也是非常有趣而且值得的，因为这不但可以让我们更了解 dbExpress，也可以从其中学习到许多东西。例如，如果是由读者您来负责开发 dbExpress 技术，那么您要如何设计这个技术才能让它与各种不同的数据库连接？如何提供一致的接口让 Delphi、C++Builder 和 Kylix 的应用程序或组件使用？如何让 dbExpress 可以在 Windows 和 Linux 平台之上使用？以及如何让未来的 dbExpress 能够在 .NET 上使用？思考这些问题都可以提升读者在软件设计和实现方面的能力，更重要的是一旦读者掌握了这些相关的概念和技术，那么在未来开发系统时读者也可以在自己的系统中使用类似的概念和技术来设计这种通用的访问结构。笔者认为这就是了解软件结构设计样例（Design Pattern）以及掌握实际开发设计样例的一个非常良好的范例。因为 Borland 不但公开了 dbExpress 的设计接口，甚至也提供了 dbExpress 驱动程序的程序源代码，让有心的程序员可以据此有效地学习其中宝贵的软件知识。

在本书的第 1 章中我们已经说明了如何设置 dbExpress 驱动程序，并且简单地说明了 dbxdrivers.ini 文件中选项的设置意义。为了本章说明方便，让我们再次列出 dbEx

press中对于MS SQL Server的设置:

```
[MSSQL]
GetDriverFunc=getSQLDriverMSSQL
LibraryName=dbexpmss.dll
VendorLib=oledb
HostName=ServerName
DataBase=Database Name
User_Name=user
Password=password
BlobSize=-1
LocaleCode=0000
MSSQL TransIsolation=ReadCommitted
OS Authentication=False
[OS Authentication]
False=0
True=1
[Multiple Transaction]
False=0
True=1
[Trim Char]
False=0
True=1
[MSSQL TransIsolation]
DirtyRead=0
ReadCommitted=1
RepeatableRead=2
```

从上面的内容中我们可以确定 dbExpress的MS SQL Server驱动程序dbexpmss.dll使用OLE DB实现它的功能, 而且dbexpmss.dll的调用进入点是getSQLDriverMSSQL函数。当dbExpress应用程序加载dbexpmss.dll之后, 应该通过GetProcAddress取得此函数的调用地址, 再直接调用此服务函数。现在让我们使用工具开始观察 dbExpress的世界, 以便让读者更加了解dbExpress。

1. 检查dbExpress驱动程序

现在先让我们检查一下 dbExpress中连接MS SQL Server和Oracle的驱动程序: dbexpmss.dll和dbexpora.dll。由于 dbExpress驱动程序提供统一的接口供Delphi/C++Builder/Kylix的dbExpress组件和技术连接使用, 因此所有 dbExpress驱动程序都支持相同的接口。只是每一个 dbExpress驱动程序分别调用不同的客户端数据库API来实际执行dbExpress要求的工作。例如dbexpmss.dll在底层使用OLE DB来实际连接MS SQL Server 2000, 而dbexpora.dll使用Oracle的OCI (Oracle Call Interface)。在说明dbExpress的实现之前先让我们实际观察一下 dbExpress驱动程序中的内容。

首先,我们可以使用 Delphi/Kylix的tdump.exe公用程序来查看 dbExpress驱动程序DLL中包含的内容,并且以此来了解 dbExpress是如何工作的。首先我们可以使用下面的指令观察dbexpmss.dll的内容:

```
tdump dbexpmss.dll
```

下面的程序代码是tdump列出的dbexpmss.dll的部分内容:

```
Description: dbExpress - MSSQL driver 32-bit dll
```

```
Section: Import
```

```
Imports from KERNEL32.DLL
```

```
...
```

```
Imports from USER32.DLL
```

```
...
```

```
Imports from OLE32.DLL
```

```
...
```

```
Imports from USER32.DLL
```

```
EnumThreadWindows
```

```
MessageBoxA
```

```
wsprintfA
```

```
Imports from OLE32.DLL
```

```
CoCreateInstance
```

```
CoInitialize
```

```
CoTaskMemFree
```

```
CoUninitialize
```

程序块2: dbexpmss.dll
输入了OLE32函数以使用OLE DB接口

```
Imports from OLEAUT32.DLL
```

```
(ord. = 200)
```

```
(ord. = 2)
```

```
(ord. = 3)
```

```
(ord. = 4)
```

```
(ord. = 5)
```

```
...
```

```
Exports from dbexpmss.dll
```

```
2 exported name(s), 2 export address(es). Ordinal base is 1.
```

```
Sorted by Name:
```

```
RVA Ord. Hint Name
```

```
-----
```

```
00017534 2 0000 ___CPPdebugHook
```

```
00001BB0 1 0000 getSQLDriverMSSQL
```

程序块1: dbexpmss.dll
输出的函数

仔细观察上面的内容我们可以发现许多有趣和重要的信息。首先请注意程序块1,我们可以看到dbexpmss.dll输出了两个函数,其中getSQLDriverMSSQL的函数正是在

前面dbxdrivers.ini文件中指定的dbExpress驱动程序进入点。其次，我们可以从另外一个输出函数 `CPPdebugHook`确定Borland的dbExpress驱动程序是使用C++语言实现的。现在请观察程序块2，dbexpmss.dll输入了OLE32.DLL中的CoCreateInstance等函数，我们可以从这一点确定dbexpmss.dll使用OLE DB接口来访问MS SQL Server 2000数据库。

现在我们再对比一下连接 Oracle数据库的dbexpora.dll驱动程序，就可以看到Borland如何在相同的接口之下使用不同的实现方式。在dbxdrivers.ini中Oracle数据库有如下的设置：

```
[Oracle]
GetDriverFunc=getSQLDriverORACLE
LibraryName=dbexpora.dll
VendorLib=OCI.DLL
BlobSize=-1
DataBase=Database Name
ErrorResourceFile=
LocaleCode=0000
Password=password
Oracle TransIsolation=ReadCommitted
User_Name=user
```

同样使用tdump观察dbexpora.dll，我们可以获得如下的输出片段：

```
Description: SQLObjects - ORACLE driver 32-bit dll
Section:          Import
Imports from KERNEL32.DLL
...
Imports from USER32.DLL
...
Exports from dbexpora.dll
  2 exported name(s), 2 export address(es).  Ordinal base is 1.
Sorted by Name:
  RVA      Ord. Hint Name
  -----
000241EC    2 0000 ____CPPdebugHook
000019FC    1 0000getSQLDriverORACLE
```

从Oracle的dbExpress驱动程序我们可以确定dbexpora.dll的进入点getSQLDriverORACLE函数和dbxdrivers.ini中指定的一样，而且dbexpora.dll也是使用C++实现的。但是请读者注意，dbexpora.dll并不像dbexpmss.dll那样输入了OLE32等DLL，因此没有使用任何COM接口。dbexpora.dll直接使用OCI.DLL提供的服务来连接Oracle数据库。因此当dbExpress应用程序使用dbexpora.dll连接Oracle数据库时，在dbexpora.dll被加载并且调用getSQLDriverORACLE函数之后，dbexpora.dll势必需

要加载OCI.DLL以连接Oracle数据库，因此如果在读者的机器中没有安装OCI.DLL或是dbexpora.dll找不到OCI.DLL，那么dbExpress都会显示图13-1所示的错误消息。



图13-1 dbExpress的Oracle驱动程序显示的错误消息

现在读者应该更能够理解为什么会出现这个错误消息了。

了解了dbExpress的初步设置并且观察了dbExpress驱动程序的内容之后，我们就准备开始探索dbExpress的实现内容了。

2. 设计dbExpress的考虑因素

设计统一的数据库访问接口是一件不容易的事情，因为这牵涉到许多设计和实现的细节。例如下面的设计和技术问题就是dbExpress开发时必须考虑的事项：

- dbExpress的设计目标是什么？
- 如何设计标准访问接口？
- 如何连接到不同的数据库？
- 如何解决不同数据库之间数据类型转换的问题？
- 如何实现数据库访问技术？
- 如何解决不同数据库之间各自特有的功能？

在下面的小节中将逐一讨论Borland的工程师在开发dbExpress技术时如何考虑、实现并且解决这些技术细节。

3. dbExpress的设计目标

dbExpress在设计时的目标就是提供一个能够跨平台的高效率数据访问引擎，而且dbExpress希望以最简单的方式来提供服务。因此dbExpress的开发目标如下：

- 高效率。
- 提供一致的接口，有效率地访问关系型数据库和存储过程。
- 允许程序员通过dbExpress访问各数据库特定的功能。
- 容易分发使用dbExpress的应用系统。
- 使用最简单的配置设置。
- 并且能够轻松地访问未来新的数据源。

dbExpress的设计也是为了解决以往BDE产生的问题。那么BDE有哪些问题呢？由于BDE是从Paradox引擎演化来的，因此当BDE必须改为访问关系型数据库时，便在其中混合了许多不同的机制并且浪费了许多资源。例如在BDE中至少保存了4份元

数据，还使用了3层的缓存内存。另外BDE也会自动产生SQL语句，并且使用复杂的配置设置。

而dbExpress在内部不提供任何缓存内存，而是直接使用客户端 DataSnap的缓存内存以避免浪费。另外dbExpress在执行时才根据需要访问元数据以避免无谓的数据访问。此外，dbExpress不会自动产生SQL语句，而只会执行客户端传递给它的SQL语句，这样设计的好处是客户端可以使用最有效率的SQL语句而不会受到dbExpress的干扰，另外DataSnap已经提供了根据数据自动产生SQL语句的能力，因此dbExpress不需要再重复提供这个功能。

4. dbExpress提供的接口

提供一致的接口以访问各种数据源是dbExpress的设计目标之一。因此Borland的dbExpress开发小组根据各种数据源的特性设计了5个不同的接口以提供一致的服务，让客户端能够通过这5个接口访问所有数据源。客户端应用程序要想使用任何数据源，一般来说需要进行下面的步骤：

- 连接数据源。
- 访问描述数据源的元数据。
- 向数据源执行客户端的命令。
- 访问和处理数据源执行命令后产生的结果数据集。

因此dbExpress提供的功能接口就和上面的步骤差不多，只是增加了许多额外的功能，让dbExpress提供的接口更为丰富。下面的表格列出并且说明了dbExpress提供的标准服务接口：

DbExpress接口	功能描述
ISQLDriver	■是dbExpress接口的进入点，负责提供ISQLConnection让客户端连接数据库并且取得其他相关的接口。所有dbExpress驱动程序必须支持此接口
ISQLConnection	■负责连接数据库，这个接口允许客户端连接数据库，并且返回连接的结果状态，或是返回ISQLCommand接口在连接数据库之后让客户端执行SQL命令
ISQLCommand	■这个接口允许客户端向后端数据库下达执行命令，例如SQL语句等。ISQLCommand在执行完毕之后可以返回其他dbExpress接口让客户端操作。例如返回ISQLCursor让客户端访问其中的数据等
ISQLCursor	■处理数据库返回的结果数据集中的数据，这个接口允许客户端使用各种访问方法来处理数据
ISQLMetaData	■处理数据库元数据的接口，这个接口让客户端可以访问和处理数据库中对对象的详细信息，例如数据表的字段以及字段的类型和长度等

在上面的标准接口中，ISQLDriver接口用于提供在DataSnap和dbExpress驱动程序之间进行绑定的功能。简单地说，ISQLDriver接口提供了外界和dbExpress之间的进入点，与处理数据没有直接的关系。

在下面的小节中，我们将简单地说明每一个dbExpress标准接口的定义并且观察

其中重要的技术信息。

5. ISQLDriver接口

dbExpress中的ISQLDriver接口主要用于进行客户端（例如 Delphi/Kylix的dbExpress组件组）以及 dbExpress驱动程序之间串联的工作。客户端通过调用dbExpress驱动程序在dbxdrivers.ini中指定的dbExpress驱动程序进入点函数就可以取得ISQLDriver接口，之后客户端就可以通过ISQLDriver接口进入dbExpress的世界。

Borland在ISQLDriver接口中定义了三个方法，这些方法的定义如下所示：

```
ISQLDriver =interface
    function GetSQLConnection (out pConn: ISQLConnection) SQLResult;stdcall;
    function SetOption (eDOption: TSQLDriverOption;
        PropValue: LongInt: SQLResult;stdcall;
    function GetOption (eDOption: TSQLDriverOption; PropValue: Pointer;
        MaxLength: SmallInt;out Length: SmallInt: SQLResult;stdcall;
end;
```

getSQLConnection方法可以返回实现ISQLConnection的接口，客户端可以通过返回的ISQLConnection接口连接数据源。从这个方法提供的功能我们就可以知道ISQLDriver是客户端和真正dbExpress功能之间的桥梁。

而SetOption和GetOption方法则是用来设置特定 dbExpress驱动程序的功能。dbExpress提供了标准接口，但不可否认的是每一个数据源都提供了不同的功能，除了让dbExpress能够提供标准的功能之外，也必须能够让客户端使用数据源特定的功能。SetOption和GetOption方法就提供了这样的能力。dbExpress通过提供标准的SetOption和GetOption方法让客户端设置或是使用数据源特定的功能。

在稍后介绍的其他dbExpress接口中读者会发现，几乎每一个dbExpress接口都会提供SetOption和GetOption这两个方法让客户端设置定制的功能。这个技巧读者也可以采用，那就是在标准接口中提供一致的方法来定义特定的或是不同的行为。

6. ISQLConnection接口

ISQLConnection接口是由刚才的ISQLDriver接口激活的，客户端通过ISQLDriver接口进入dbExpress的世界。而ISQLConnection接口的功能就是提供客户端和数据源之间的连接，并且让客户端能够通过ISQLConnection接口取得其他dbExpress标准接口以操纵和处理数据源。

当客户端通过ISQLDriver接口取得了ISQLConnection接口之后，就可以调用其中的connect方法连接数据源，并且传入数据源的服务器名称、登录用户名和密码，disconnect则可以切断客户端和数据源的连接。如果客户端准备执行SQL语句，那么可以调用getSQLCommand以取得ISQLCommand接口。如果客户端需要取得数据源中的元数据，那么getSQLMetaData方法可以让客户端取得ISQLMetaData接口，再通

过ISQLMetaData接口访问数据源中的任何元数据。

```
ISQLConnection interface
function connect (ServerName: PChar; UserName: PChar;
                  Password: PChar: SQLResult;stdcall;
function disconnect: SQLResult;stdcall;
function getSQLCommand (out pComm: ISQLCommand: SQLResult;stdcall;
function getSQLMetaData (out pMetaData: ISQLMetaData: SQLResult;stdcall;
function SetOption (eConnectOption: TSQLConnectionOption;
                   lValue: LongInt: SQLResult;stdcall;
function GetOption (eOption: TSQLConnectionOption; PropValue: Pointer;
                   MaxLength: SmallInt;out Length: SmallInt: SQLResult;stdcall;
function beginTransaction (TranID: LongWord: SQLResult;stdcall;
function commit (TranID: LongWord: SQLResult;stdcall;
function rollback (TranID: LongWord: SQLResult;stdcall;
function getErrorMessage (Error: PChar: SQLResult; overload;stdcall;
function getErrorMessageLen (out ErrorLen: SmallInt: SQLResult;stdcall;
end;
```

当客户端需要进行事务管理以便把数据修改回数据源时，可以调用 `beginTransaction` 进入事务管理模式，执行完修改数据的工作后根据执行结果来决定调用 `commit` 还是 `rollback` 方法。请读者注意，`beginTransaction`、`commit` 和 `rollback` 都接受一个事务ID作为处理的目标，这和前面章节介绍事务管理时使用事务ID是一样的道理。由于 `dbExpress` 支持事务ID和嵌套式事务，因此 `dbExpress` 组件组才需要使用 `TTransactionDesc` 来控制事务ID。

最后请读者注意，在 `ISQLConnection` 接口中同样提供了 `SetOption` 和 `GetOption` 方法以便允许客户端定制和设置连接的特定功能和行为。

7. ISQLCommand接口

`dbExpress` 的 `ISQLCommand` 接口的主要功能是向客户端提供执行 SQL 语句的能力。由于 SQL 语句可能包含动态参数，因此在 `ISQLCommand` 接口中也提供了设置动态参数的方法，让客户端能够设置动态参数的类型、精确度等特性。

由于 SQL 语句可分为会返回结果数据集的 SQL 语句以及不返回结果数据集的 SQL 语句，因此 `ISQLCommand` 接口执行了 SQL 语句之后对于会返回结果数据集的 SQL 语句提供了 `ISQLCursor` 接口，让客户端可以用这个接口操作结果数据集中的数据。

```
ISQLCommand interface
function SetOption (
    eSqlCommandOption: TSQLCommandOption;
    ulValue: Integer: SQLResult; stdcall;
function GetOption (eSqlCommandOption: TSQLCommandOption;
    PropValue: Pointer;
    MaxLength: SmallInt;out Length: SmallInt: SQLResult;stdcall;
```

```

function setParameter (
    ulParameter: Word ;
    ulChildPos: Word ;
    eParamType: TSTMTParamType ;
    uLogType: Word;
    uSubType: Word;
    iPrecision: Integer;
    iScale: Integer;
    Length: LongWord ;
    pBuffer: Pointer;
    lInd: Integer): SQLResult;stdcall;
function getParameter (ParameterNumber: Word; ulChildPos: Word; Value:
    Pointer;
    Length: Integer; var IsBlank: Integer): SQLResult;stdcall;
function prepare (SQL: PChar; ParamCount: Word): SQLResult;stdcall;
function execute (var Cursor: ISQLCursor): SQLResult;stdcall;
function executeImmediate (SQL: PChar; var Cursor: ISQLCursor ) : SQLResult; stdcall;
function getNextCursor (var Cursor: ISQLCursor): SQLResult;stdcall;
function getRowsAffected (var Rows: LongWord): SQLResult;stdcall;
function close: SQLResult;stdcall;
function getErrorMessage (Error: PChar): SQLResult; overload; stdcall;
function getErrorMessageLen (out ErrorLen: SmallInt): SQLResult;stdcall;
end;

```

在上面的 ISQLCommand 接口中，当客户端通过调用 ISQLConnection.getSQLCommand 取得 ISQLCommand 接口之后，就可以调用其中的 execute 方法执行 SQL 语句。如果 SQL 语句中使用了动态参数，就需要先调用 setParameter 设置动态参数的特性值。

对于需要执行数次的 SQL 语句，客户端可以先调用 prepare 方法要求后端数据源先编译并且准备 SQL 语句，这样可以得到较好的性能。对于只执行一次或是属于 DDL 类的 SQL 语句，那么客户端可以直接使用 executeImmediate 方法。此外，在 ISQLCommand 接口中还提供了 getNextCursor 方法允许客户端使用分段的方式访问数据。

最后，ISQLCommand 接口也与 ISQLConnection 和 ISQLDriver 接口一样提供了 SetOption 和 GetOption 方法以帮助客户端执行定制的工作。

8. ISQLCursor 接口

ISQLCursor 接口用来控制和管理由 ISQLCommand 接口返回的结果数据集。客户端通过 ISQLCursor 接口可以访问结果数据集中的数据以及每一个字段中的值。由于客户端可使用动态 SQL 语句来访问数据，因此在许多情形中客户端需要同时通过稍后介绍的 ISQLMetaData 接口以及 ISQLCursor 接口来访问字段的数据。客户端通过 ISQLMetaData 接口得知字段的类型和精确度等信息，再通过 ISQLCursor 接口中不同

种类的方法来访问字段值。下面是 ISQLCursor 接口的定义：

```
ISQLCursor = interface
    function SetOption (eOption: TSQLCursorOption;
        PropValue: LongInt; SQLResult; stdcall;
    function GetOption (eOption: TSQLCursorOption; PropValue: Pointer;
        MaxLength: SmallInt; out Length: SmallInt) : SQLResult; stdcall;
    function GetErrorMessage (Error: PChar: SQLResult; overload; stdcall;
    function GetErrorMessageLen (out ErrorLen: SmallInt: SQLResult; stdcall;
    function GetColumnCount (var pColumns: Word: SQLResult; stdcall;
    function GetColumnNameLength (
        ColumnNumber: Word;
        var pLen: Word : SQLResult; stdcall;
    function GetColumnName (ColumnNumber: Word; pColumnName: PChar ) : SQLResult; stdcall;
    function GetColumnType (ColumnNumber: Word; var puType: Word;
        var puSubType: Word: SQLResult; stdcall;
    function GetColumnLength (ColumnNumber: Word; var pLength: LongWord:
        SQLResult; stdcall;
    function GetColumnPrecision (ColumnNumber: Word;
        var piPrecision: SmallInt: SQLResult; stdcall;
    function GetColumnScale (ColumnNumber: Word; var piScale: SmallInt:
        SQLResult; stdcall;
    function IsNullable (ColumnNumber: Word; var Nullable: LongBool: SQLResult;
        stdcall;
    function IsAutoIncrement (ColumnNumber: Word; var AutoIncr: LongBool:
        SQLResult; stdcall;
    function IsReadOnly (ColumnNumber: Word; var ReadOnly: LongBool) : SQLResult;
        stdcall;
    function IsSearchable (ColumnNumber: Word; var Searchable: LongBool:
        SQLResult; stdcall;
    function IsBlobSizeExact (ColumnNumber: Word; var IsExact: LongBool:
        SQLResult; stdcall;
    function Next: SQLResult; stdcall;
    function GetString (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetShort (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetLong (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetDouble (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetBcd (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetTimeStamp (ColumnNumber: Word; Value: Pointer;
        var IsBlank: LongBool: SQLResult; stdcall;
    function GetTime (ColumnNumber: Word; Value: Pointer;
```

```

var IsBlank: LongBool: SQLResult;stdcall;
function getDate (ColumnNumber: Word; Value: Pointer;
var IsBlank: LongBool: SQLResult;stdcall;
function getBytes (ColumnNumber: Word; Value: Pointer;
var IsBlank: LongBool: SQLResult;stdcall;
function getBlobSize (ColumnNumber: Word; var Length: LongWord;
var IsBlank: LongBool: SQLResult;stdcall;
function getBlob (ColumnNumber: Word; Value: Pointer;
var IsBlank: LongBool; Length: LongWord; SQLResult;stdcall;
end;

```

请读者注意，在 ISQLCursor 接口中定义了许多方法，每一个方法都是为了访问不同类型的字段值。例如，对于整数类型的字段，客户端可使用 `getShort` 或是 `getLong`。对于字符串类型则使用 `getSting` 方法。此外也提供了许多 `IsXXX` 方法来判断结果数据集中的字段是否属于特定的类型。

在使用 ISQLCursor 接口访问结果数据集中的数据时，客户端可以先调用 `getColumnCount` 取得目前结果数据集中的字段数量，再进入一个循环调用 `getColumnType` 判断每一个字段的类型，再根据字段类型决定调用哪一个 `getXXX` 方法取得字段值。在处理完毕之后调用 `next` 方法继续处理下一个记录。

最后，ISQLCursor 接口也与 ISQLCommand、ISQLConnection 和 ISQLDriver 接口一样提供了 `SetOption` 和 `GetOption` 方法以帮助客户端执行定制的工作。

9. ISQLMetaData 接口

最后介绍的 ISQLMetaData 接口负责使客户端能够访问数据源中的元数据信息。在第9章中本书详细讨论了元数据，因此现在读者应该很清楚地知道什么是元数据。既然 ISQLMetaData 接口负责提供元数据信息，那么在这个接口中当然应该存在着许多方法，客户端可以调用它们来取得数据源的数据表信息、索引信息以及存储过程等信息。下面是 ISQLMetaData 接口的定义：

```

ISQLMetaData interface
function SetOption (eOption: TSQLMetaDataOption;
    PropValue: LongInt: SQLResult;stdcall;
function GetOption (eOption: TSQLMetaDataOption; PropValue: Pointer;
    MaxLength: SmallInt; out Length: SmallInt: SQLResult;
    stdcall;
function getObjectList (eObjType: TSQLObjectType; out Cursor: ISQLCursor;
    SQLResult;stdcall;
function getTables (TableName: PChar; TableType: LongWord;
    out Cursor: ISQLCursor: SQLResult;stdcall;
function getProcedures (ProcedureName: PChar; ProcType: LongWord;
    out Cursor: ISQLCursor: SQLResult;stdcall;
function getColumns (TableName: PChar; ColumnName: PChar;

```

```

ColType: LongWord; out Cursor: ISQLCursor; SQLResult;
stdcall;
function getProcedureParams (ProcName: PChar; ParamName: PChar;
    out Cursor: ISQLCursor; SQLResult; stdcall;
function getIndices (TableName: PChar; IndexType: LongWord;
    out Cursor: ISQLCursor; SQLResult; stdcall;
function getErrorMessage (Error: PChar; SQLResult; overload; stdcall;
function getErrorMessageLen (out ErrorLen: SmallInt; SQLResult; stdcall;
end;

```

ISQLMetaData接口可以通过 ISQLConnection接口中的 getSQLMetaData方法取得，之后客户端便可使用 ISQLMetaData接口提供的服务。例如 getTables可取得数据源中指定类型的数据表。getIndices则可以让客户端取得数据源中特定数据表的索引信息。如果读者还记得第9章讨论的内容，那么可以发现 ISQLMetaData接口的许多方法与dbExpress组件组提供的方法非常类似，简单地说 dbExpress组件组几乎是把 ISQLMetaData接口中的方法以对应的方式提供给 Delphi/Kylix程序员使用，只是dbExpress组件组提供了较为简洁的调用方式，并且提供了数据类型转换的工作。例如，在第9章中本书讨论的 GetTables方法可取得数据源中的数据表信息，而在dbExpress组件组中则是直接使用 ISQLMetaData接口的getTables方法，如下所示：

```

TblType := GetTableScope(GetInternalConnection.FTableScope);
Status := GetInternalConnection.FSQLMetaData.getTables
    WildCard, TblType, FSQLCursor

```

上面的 TblType和 WildCard具有如下的声明：

```

TblType: LongWord;
WildCard: PChar;

```

从上面的程序代码可以看到，dbExpress组件组直接使用了 ISQLMetaData定义的数据类型（以 C/C++为主的数据类型）。当dbExpress组件组成功地使用了 dbExpress标准接口之后会主动转换为 Object Pascal的数据类型再返回给客户端。

上面介绍的就是 dbExpress定义的标准接口，从上面的讨论中读者也应该会发现这些dbExpress标准接口之间都有一定的交互关系，我们可以使用图 13 2来清楚地描述dbExpress标准接口之间的关系以及交互的情形。

不管后端数据源到底是哪种关系型数据库或是数据源，dbExpress都能够以相同的dbExpress标准接口来提供客户端和后端数据源之间的数据访问工作，因此 Delphi/Kylix的dbExpress组件组只需要使用这些 dbExpress标准接口即可控制各种数据源。同样，不管后端数据源是什么，客户端的 Delphi/Kylix应用程序都能够使用一致的dbExpress组件组访问和处理数据，如图 13 3所示。

这种设计允许 Delphi/Kylix提供最大的弹性以及快速的开发能力。

从前面讨论的dbExpress标准接口中读者可能会发现，dbExpress接口事实上定义

得非常简单且有效率，但是缺少了许多数据库应用系统需要的数据访问功能。而dbExpress缺乏的这些功能就由客户端的DataSnap技术来弥补，从Delphi 6之后Borland就清楚地分离了数据访问引擎的核心功能以及客户端需要的丰富的数据操作功能。这样的设计可以让Borland的dbExpress开发小组和DataSnap开发小组能够独立地进行开发，也同时实现了尽量让数据访问引擎简单、体积小又有效率，并且让客户端的数据处理能力丰富又好用的双重目的。

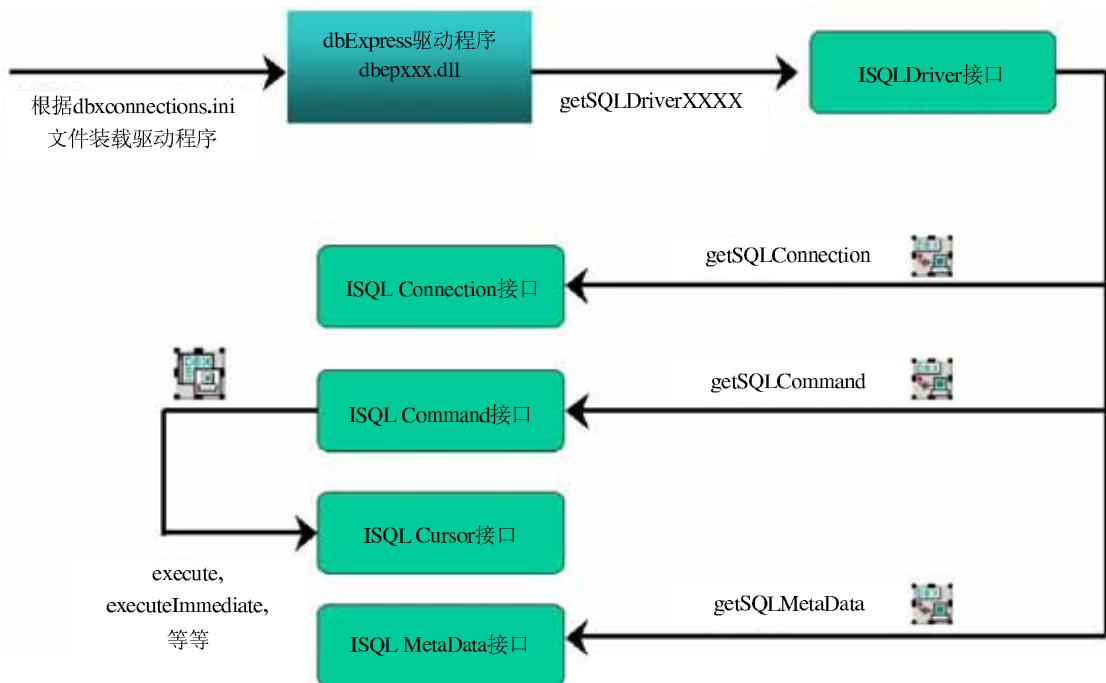


图13-2 dbExpress和接口的关系流程

13.2 模拟dbExpress的工作流程

到了这里读者应该对于dbExpress技术和驱动程序有了一定的了解，也能够了解Borland在设计和开发dbExpress时考虑的事项有哪些。读者可以在设计自己的应用系统时使用这些dbExpress知识。但是笔者相信许多读者一定也觉得虽然对于dbExpress有了比较深入的了解和掌握，但是似乎对于dbExpress是如何实现的仍然有点模糊。为了让读者能够真正掌握如何从dbExpress的定义标准实现最终的dbExpress驱动程序，让我们使用一个简单的范例来说明。这个范例将模拟dbexpmss.dll的开发，让读者了解如何使用Delphi/Kylix来开发dbExpress驱动程序。当然本小节展示的范例不是完整的dbExpress驱动程序，这个范例的目的是让读者了解如何从别人定义的标准来

实现特定的软件技术。

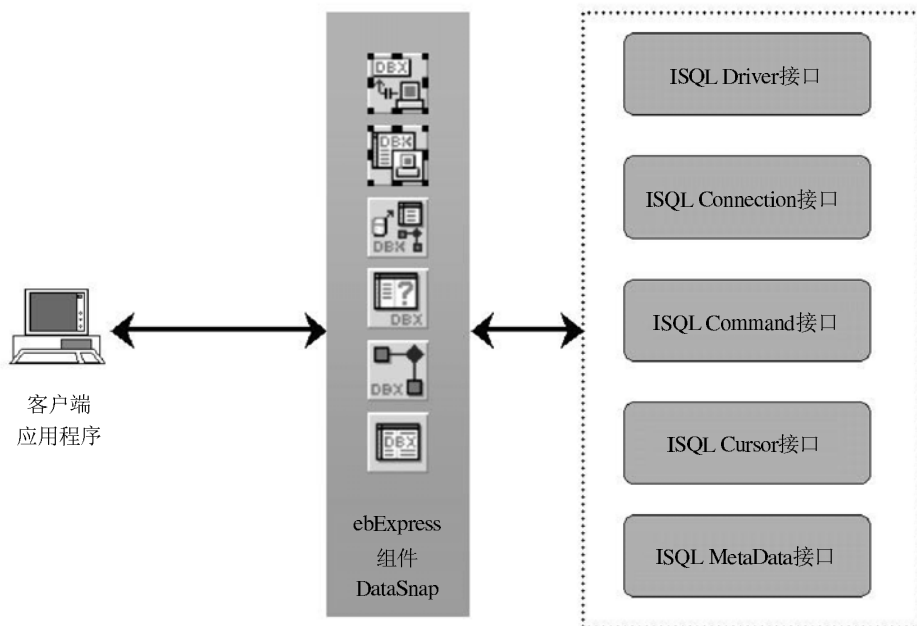


图13-3 dbExpress通过统一的接口和Delphi组件向客户端应用程序提供一致的访问方式

因此本小节的讨论只限于让读者了解如何使用 Object Pascal实现dbExpress接口，以及如何在 Object Pascal实现类中调用特定厂商的客户端 API来连接和访问数据源。如果读者要学习如何完整地实现 dbExpress驱动程序，那么 Borland公布了完整的dbExpress For MySQL驱动程序，读者可以在阅读完本章并且了解了 dbExpress开发的基本概念之后到网上下载完整的源代码来研究。

在下面的小节中将使用实际的范例介绍如何使用 Object Pascal来实现dbExpress的标准接口。

1. 建立DLL项目，输出dbExpress进入点函数

要使用Delphi/Kylix开发dbExpress驱动程序并不困难，因为 Delphi/Kylix已经提供了Object Pascal程序代码的dbExpress标准接口定义。这些dbExpress标准接口定义存在于DBXpress程序单元中，开发者只需要使用这个程序单元就可以开始开发dbExpress驱动程序。唯一比较麻烦的是一般关系型数据库的客户端 API大都是以C/C++编写的，因此如果读者想要使用 Delphi/Kylix来开发dbExpress驱动程序，那么必须把各种关系型数据库的客户端 API适当地转换为Object Pascal的程序单元定义，再进行调用。如果读者嫌麻烦，那么可以先使用 C++Builder或是C/C++版本的Kylix实现一个wrapper类调用各种关系型数据库的客户端 API，再使用Delphi/Kylix调用C++Builder或是C/C++版本Kylix实现的wrapper类即可。不过即使读者不需要使用Delphi/Kylix开发dbExpress驱动程序，看看本小节如何使用 Object Pascal实现

dbExpress标准接口也可以学习到许多技巧。因此读者可以抱着欣赏的心情来阅读本小节的内容，或是跳过本小节讨论的主题。

要使用 Delphi/Kylix 开发 dbExpress 驱动程序，首先我们可以在 Delphi/Kylix 中点击 DLL Wizard 图标让 Delphi/Kylix 为我们建立一个 DLL 项目，这当然是因为 dbExpress 驱动程序必须是 DLL 类型的文件（见图 13-4）。

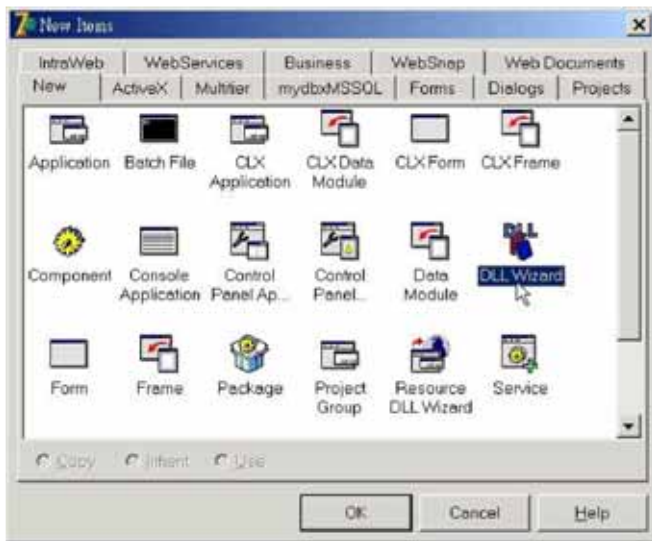


图13-4 在Delphi中建立DLL项目以准备开发dbExpress驱动程序

在 Delphi/Kylix 产生了 DLL 项目之后，接下来的一步就是输出 dbExpress 驱动程序的进入点函数，这个函数也是在 dbxdrivers.ini 中声明的进入点函数，例如：

```
[MyDemoDBXDriver]
GetDriverFunc= getMyDBXSQLDriver
LibraryName=dbexpDemoDriver.dll
```

那么我们就需要在 DLL 项目中声明输出函数 `getMyDBXSQLDriver`，而 `getMyDBXSQLDriver` 函数需要返回 dbExpress 标准接口 `ISQLDriver`，因此我们可以实现如下的程序代码：

```
exports
    getMyDBXSQLDriver;

Function getMyDBXSQLDriver (VendorLib, SResourceFile : PChar;
    out pDriver) : SQLResult; StdCall;
Begin
    ISQLDriver (pDriver) := TSQLDriverImpl.Create;
    Result := SQL_SUCCESS;
End;
```

上面的程序代码便是 dbExpress 驱动程序的进入点函数，它指定 `getMyDBX`

SQLDriver是DLL的输出函数，也是此dbExpress驱动程序的进入点函数。它的实现很简单，它建立了一个 TSQLDriverImpl对象，并且把这个对象指定给返回参数 pDriver。由于返回参数 pDriver是存储dbExpress中ISQLDriver接口的返回结果，因此我们可以推知TSQLDriverImpl一定是一个实现ISQLDriver接口的类。

2. 实现ISQLDriver接口

刚才介绍的 TSQLDriverImpl声明为从 TInterfacedObject类继承下来并且实现了 ISQLDriver接口。由于 TSQLDriverImpl是从 TInterfacedObject类继承下来的，因此它具备了 TInterfacedObject类提供的基本接口管理以及引用计数值管理的功能。此外 TSQLDriverImpl也必须实现ISQLDriver定义的三个方法。

Type

```
TSQLDriverImpl Class (TInterfacedObject,ISQLDriver)
function GetSQLConnection (out pConn: ISQLConnection) SQLResult;stdcall;
function SetOption (eDOption: TSQLDriverOption;
    PropValue: LongInt: SQLResult;stdcall;
function GetOption (eDOption: TSQLDriverOption; PropValue: Pointer;
    MaxLength: SmallInt; out Length: SmallInt: SQLResult;
    stdcall;
End;
```

在这个范例dbExpress驱动程序中，我们并不讨论如何通过 GetOption/SetOption 设置特定数据源的特殊功能，让我们关注 TSQLDriverImpl如何通过 getSQLConnection方法将ISQLConnection接口提供给客户端。

从下面的实现程序代码可以看到， TSQLDriverImpl的getSQLConnection方法只是建立一个TSQLConnectionImpl对象，再把此对象指定给返回参数 pConn，并没有再执行任何特别的工作。这是因为 ISQLDriver接口只是提供客户端和dbExpress标准接口之间的连接工作，因此没有什么复杂的实现工作。 ISQLDriver唯一比较复杂的地方应该是确定这个dbExpress驱动程序能够加载连接的关系型数据库客户端 API函数库，这是这个范例没有给出的内容，在读者实际实现 ISQLDriver接口时需要提供这个功能。

```
function TSQLDriverImpl.GetOption (eDOption: TSQLDriverOption;
    PropValue: Pointer; MaxLength: SmallInt;
    out Length: SmallInt: SQLResult;
begin
    Result := DBXERR_NOTSUPPORTED;
end;

function TSQLDriverImpl.getSQLConnection (
    out pConn: ISQLConnection) SQLResult;
begin
```

```

    pConn := TSQLConnectionImpl.Create;
    Result := SQL_SUCCESS;
end;

function TSQLDriverImpl.SetOption (eOption: TSQLDriverOption;
    PropValue: Integer): SQLResult;
begin
    if (eOption = eDrvRestrict) then
        Result := SQL_SUCCESS
    else
        Result := SQLResul(DBXERR_NOTSUPPORTED);
end;

```

由于TSQLDriverImpl的getSQLConnection返回一个TSQLConnectionImpl对象，因此我们也可以推知TSQLConnectionImpl类一定实现了ISQLConnection接口。

3. 实现ISQLConnection接口

刚才讨论的TSQLDriverImpl.getSQLConnection方法返回了TSQLConnectionImpl对象，因此TSQLConnectionImpl类是从TInterfacedObject类继承下来的并且实现了ISQLConnection接口。

TSQLConnectionImpl提供了与数据源之间的连接，因此TSQLConnectionImpl类开始需要调用实际数据源的客户端API来与数据源进行交互了，在TSQLConnectionImpl类中也因此开始出现特定实现的程序代码。由于客户端与数据源进行连接时许多数据源客户端API都要求客户端先配置资源，因此通常在TSQLConnectionImpl的构造函数中也需要配置资源和内存以调用数据源的客户端API。

```

TSQLConnectionImpl Class (TInterfacedObject, ISQLConnection)
function connect (ServerName: PChar; UserName: PChar;
    Password: PChar): SQLResult; stdcall;
function disconnect: SQLResult; stdcall;
function getSQLCommand (out pComm: ISQLCommand): SQLResult; stdcall;
function getSQLMetaData (out pMetaData: ISQLMetaData): SQLResult; stdcall;
function SetOption (eConnectOption: TSQLConnectionOption;
    lValue: LongInt): SQLResult; stdcall;
function GetOption (eOption: TSQLConnectionOption; PropValue: Pointer;
    MaxLength: SmallInt; out Length: SmallInt): SQLResult; stdcall;
function beginTransaction (TranID: LongWord): SQLResult; stdcall;
function commit (TranID: LongWord): SQLResult; stdcall;
function rollback (TranID: LongWord): SQLResult; stdcall;
function getErrorMessage (Error: PChar): SQLResult; overload; stdcall;
function getErrorMessageLen (out ErrorLen: SmallInt): SQLResult; stdcall;
Protected
    DBLoginRec : PLoginRec;

```

```

DBProc          : PDBProcess;
HostName        : String;
DatabaseName    : String;
...

procedure AllocateResources;
procedure DestroyResources;

Public
    Constructor Create; override;
    Destructor Destroy; override;

End;

```

因此在下面的 `TSQLConnectionImpl` 实现中，`TSQLConnectionImpl` 在它的构造函数中调用了 `AllocateResources` 方法来配置客户端的资源以及内存，而 `AllocateResources` 方法则实际调用了 Oracle 的 `Call Interface API` 来配置客户端的资源。

```

{ TSQLConnectionImpl }

ConstructorTSQLConnectionImpl.Create;
Begin
    InheritedCreate;
    AllocateResources;
End;

DestructorTSQLConnectionImpl.Create;
Begin
    DestroyResources;
    Inherited Destroy;
End;

ProcedureTSQLConnectionImpl.AllocateResources;
Begin
// Initialize OCI and environment handle
    status = OCIInitialize(OCI_THREADED | OCI_OBJECTS,(dvoid *)0, 0, 0, 0);

    status = OCIEEnvInit((OCIEEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
(dvoid **) 0 );

    // Allocate service, server, authentication and error handles
    status = OCIHandleAll6c(dvoid *)envhp, (dvoid **) &svchp,
OCI_HTYPE_SVCCTX,(size_t) 0, (dvoid **) 0 );

    status = OCIHandleAll6c(dvoid *)envhp, (dvoid **) &srvhp,
OCI_HTYPE_SERVER,(size_t) 0, (dvoid **) 0 );

```

```

        status = OCIHandleAll((dvoid *) envhp, (dvoid **) &usrhp,
        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);

        status = OCIHandleAll((dvoid *) envhp, (dvoid **) &errhp,
        OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
    End;

Procedure TSQLConnectionImpl.DestroyResources;
Begin
    DestroyResources;
    Inherited Destroy;
End;

function TSQLConnectionImpl.beginTransaction (TranID: LongWord): SQLResult;
begin
    Result := DBXERR_NOTSUPPORTED;
end;

function TSQLConnectionImpl.commit (TranID: LongWord): SQLResult;
begin
    Result := DBXERR_NOTSUPPORTED;
end;

function TSQLConnectionImpl.connect (ServerName, Username, Password: PChar):
    SQLResult;
begin
    // Connect and set server context in the service context
    status = OCIServerAttach (srvhp, errhp, (text *) pszDBName, strlen(pszDBName), 0);

    status = OCIAttrSet (dvoid *) svchp, OCI_HTYPE_SVCCTX,
    srvhp, (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

    // Set username and password in the authentication handle
    status = OCIAttrSet ((dvoid *) usrhp, OCI_HTYPE_SESSION, (dvoid *) pszUserId,
    (ub4) strlen (pszUserId), OCI_ATTR_USERNAME, errhp);

    status = OCIAttrSet ((dvoid *) usrhp, OCI_HTYPE_SESSION,
        (dvoid *) pszPwd, (ub4) strlen (pszPwd), OCI_ATTR_PASSWORD, errhp);

    // Authenticate and set authentication context in service context
    status = OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, (ub4)
    OCI_DEFAULT);

```



```

status = OCIAttrS((dvoid *) svchp, OCI_HTYPE_SVCCTX,
    (dvoid *) usrh, (ub4) 0, (ub4) OCI_ATTR_SESSION,
    (OCIError *) errhp);
DebugOut (Self, 'Returning: '+IntToS(Result));
Result := SQL_SUCCESS;
end;

function TSQLConnectionImpl.disconnect: SQLResult;
Var Res, Row, ResCode : Integer;
begin
    Result := SQL_SUCCESS;
    DebugOut (Self, 'Returning: '+IntToStr(Result));
end;

function TSQLConnectionImpl.getErrorMessage (Error: PChar): SQLResult;
begin
    Result := SQL_SUCCESS; { fake }
end;

function TSQLConnectionImpl.getErrorMessageLen (
    out ErrorLen: SmallInt): SQLResult;
begin
    ErrorLen := 0;
    Result := SQL_SUCCESS; { fake }
end;

function TSQLConnectionImpl.GetOption (eDOption: TSQLConnectionOption;
    PropValue: Pointer; MaxLength: SmallInt;
    out Length: SmallInt): SQLResult;
begin
    Result := SQL_SUCCESS;
end;

function TSQLConnectionImpl.getSQLCommand (
    out pComm: ISQLCommand ): SQLResult;
begin
    pComm := TSQLCommandImpl.Create (Self);
    Result := SQL_SUCCESS;
end;

function TSQLConnectionImpl.getSQLMetaData (
    out pMetaData: ISQLMetaData ): SQLResult;

```

```

begin
    pMetaData := TSQLMetaDataImpl.Create;
    Result := SQL_SUCCESS;
end;

function TSQLConnectionImpl.rollback (TranID: LongWord): SQLResult;
begin
    Result := DBXERR_NOTSUPPORTED;
end;

function TSQLConnectionImpl.SetOption (eConnectOption: TSQLConnectionOption;
    lValue: Integer): SQLResult;
begin
    Result := SQL_SUCCESS; { fake }
end;

```

TSQLConnectionImpl类中实际负责与数据源连接的 `connect` 方法则使用了 OCI 的 API 建立连接会话，并且开启与数据源的连接。由于 TSQLConnectionImpl 类的 `getSQLCommand` 方法会返回 ISQLCommand 接口让客户端执行 SQL 语句，因此在上面的实现程序代码中同样使用了类似 ISQLDriver 接口串连 ISQLConnection 接口的技巧，建立一个 TSQLCommandImpl 对象并且返回此对象。因此我们可以得知 TSQLCommandImpl 是一个实现 ISQLCommand 接口的类。

4. 实现 ISQLCommand 接口

最后讨论的 TSQLCommandImpl 与 TSQLConnectionImpl 和 TSQLDriverImpl 类一样也是从 TInterfacedObject 继承下来的并且实现了 ISQLCommand 接口。我们在这里不再详细列出 TSQLCommandImpl 实现的 ISQLCommand 接口定义。

```

TSQLCommandImpl class (TInterfacedObject, ISQLCommand)
...
protected
    OwnerConnection : TSQLConnectionImpl;
    SQLToExecute      : String;
    SQLExecuted       : Boolean;

procedure AllocateStatementHandles;
...
constructor Create (AOwnerConnection : TSQLConnectionImpl);
...
end;

```

由于 TSQLCommandImpl 类向客户端提供执行 SQL 语句的功能，并且在执行完毕之后返回 ISQLCursor 接口，因此 TSQLCommandImpl 在构造函数中必须先配置执行 SQL 语句所需的资源。在 TSQLCommandImpl 的构造函数中调用 `AllocateStatement`

Handles以配置必要的客户端资源。而在 `AllocateStatementHandles` 方法中则需要调用数据源的客户端API来配置资源，例如下面的实现程序代码调用了OCI的OCIHandleAlloc方法来配置语句句柄。

最后，真正执行SQL语句的`execute`方法调用了OCI的OCIStmtExecute方法来执行客户端的SQL语句。在执行完毕之后，`execute`方法需要返回用于处理数据的ISQLCursor接口，因此`execute`方法建立了一个实现ISQLCursor接口的TSQLCursorImpl对象并且返回给客户端。

```

constructor TSQLCommandImpl.Create (AOwnerConnection: TSQLConnectionImpl
begin
    inherited Create;
    OwnerConnection := AOwnerConnection;
    AllocateStatementHandles;
end;

Procedure TSQLCommandImpl.AllocateStatementHandles;
Begin
    // Allocate a statement handle
    status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &stmhp,
        (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid *) 0);

    // Allocate a describe handle
    status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschp,
        (ub4) OCI_HTYPE_DESCRIBE, (size_t) 0, (dvoid *) 0);
End;

function TSQLCommandImpl.execute (var Cursor: ISQLCursor) return SQLResult;
Begin
    status = OCIStmtExecute((dvoid *) envhp, stmhp, errhp, (ub4) iter, (ub4) 0,
        (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_COMMIT_ON_SUCCESS);

    ...
    Cursor := TSQLCursorImpl.Create(self);
    Result := SQL_SUCCESS;
End;

```

从上面的讨论中读者可以了解到，使用 Object Pascal实现dbExpress标准接口是非常容易的，只需要从 TInterfacedObject继承下来并且实现特定的 dbExpress标准接口即可。此外上面的程序代码在实现通过一个 dbExpress接口串联另外一个dbExpress接口时，例如从ISQLConnection的getSQLCommand方法取得ISQLCommand接口，使用的技巧就是返回实现dbExpress接口的类。读者在自己的程序代码中也可以使用这种技巧。

现在，读者应该对于如何使用 Delphi/Kylix 开发 dbExpress 驱动程序有了基本的了解。如果读者还想继续深入了解更多的实现细节，笔者建议读者下载 Borland 提供的 dbExpress 驱动程序的开放源代码以研究更多的实现技巧。

13.3 有关 dbExpress 实现的异同

虽然每一个 dbExpress 驱动程序都是遵照 Borland 定义的 dbExpress 标准接口实现的，但是不同的 dbExpress 驱动程序在访问不同的数据源时仍然可能有不一样的执行行为，例如前面章节讨论的元数据处理方式等，这些是读者在使用 dbExpress 时需要了解的。

读者还需要知道，即使是访问相同的数据源，dbExpress 驱动程序的执行行为与其他访问引擎也可能会有所不同。例如 dbExpress 和 BDE 在访问数据源、处理数据以及使用缓冲存储器方面有非常大的差异。让我们以一个实际的例子再详细说明一下。

在 Delphi 7 中 Borland 提供了访问 MS SQL Server 2000 的 dbExpress 驱动程序，而 dbExpress For MS SQL Server 是使用 OLE DB 访问数据库的，而同样在 Delphi 中的 dbGo (ADO) 也是通过 OLE DB 访问 MS SQL Server 的。虽然这两者都可以访问 MS SQL Server，但是它们在许多处理数据的方式上却有可能不同。例如在 OLE DB 中提供了 RowSet 接口让客户端访问数据集，在 dbExpress For MS SQL Server 中使用的则是通过使用 IMultipleResultset 接口的 GetResult () 方法来取得 RowSet 接口：

```
iRowSet = IMultipleResultset->GetResult
```

但是 ADO 却可能是直接使用 Command 接口来取得 RowSet：

```
iRowSet = Command->Execute;
```

造成这样的现象是因为 OLE DB 提供了许多不同的方式来完成相同的工作，当然不同的程序员会使用不同的方式来实现数据访问技术。又例如 ADO 提供了多种不同的游标访问方式，而 dbExpress 由于设计的概念而只提供客户游标。此外，ADO 会在内部缓存许多元数据信息，而 dbExpress 却只在需要时才访问元数据。这些都是 dbExpress For MS SQL Server 与 ADO 不一样的地方，是读者或程序员在使用这两个不同的数据访问引擎时需要了解的。

前面举的例子是对于 MS SQL Server 而言，对于其他数据源也是一样的，dbExpress 可能会和其他数据访问引擎不一样。下面列出了读者在阅读完本小节之后应该知道的事项：

- 程序员在使用 dbExpress 时最好先使用 TSQLMonitor 等组件观察一下 dbExpress 驱动程序的执行行为。
- 访问相同的数据源时不同的数据访问引擎会有不同的表现，因此可能会呈现不同的执行行为，bug 也有可能不一样。

- dbExpress仍然在快速的进步之中，如果程序员经常更换 dbExpress驱动程序，那么最好有一个固定的 dbExpress测试标准，以保证不同版本的 dbExpress不会造成应用系统的执行行为改变。

13.4 dbExpress未来的实现开发

Borland已经在Delphi 7和BDE End Of Life的文件中很清楚地说明，BDE现在已经进入结束的状态，除了与纯文件数据库（例如 Paradox和dBase）进行连接之外，Delphi/C++Builder/Kylix的用户应该开始使用 dbExpress，而不要再使用 BDE。dbExpress的优点在前面的章节中已经说明得很清楚了。而另外一个使用 dbExpress的重要原因Borland现在已经开始进行把dbExpress移植到.NET平台的工作了。

Borland计划把dbExpress移植到.NET平台，让dbExpress同时支持原始Windows应用程序开发、Linux应用程序开发，以及明年 Borland在.NET平台上的开发，那就是Borland的.NET开发工具Galileo。目前Microsoft的.NET开发是使用ADO.NET技术访问数据库，而 ADO.NET现在支持的数据库是相当有限的，包括 MS SQL Server、Access和Oracle。而dbExpress支持更多的数据库并且提供了更好的性能，因此当dbExpress能够支持.NET平台时，.NET的用户就可以选择使用更多的数据库，例如MySQL和InterBase等。

由于在.NET平台上是使用 ADO.NET提供访问数据的能力，因此 Borland在.NET平台上提供dbExpress功能的可能实现方式是融合 dbExpress的标准接口和ADO.NET的接口。在读者继续阅读之前，笔者仍然建议读者先停下来想想如果是读者负责把dbExpress移植到.NET平台，那么读者会如何设计和实现。

事实上Borland可以用两种方式来设计 .NET平台上的dbExpress访问：

- 快速访问机制。
- 兼容访问机制。

所谓快速访问机制是指 Borland可以自行开发 Galileo使用的.NET数据访问组件。而这些数据访问组件使用 .NET的InterOp能力直接调用dbExpress驱动程序，就像现在Delphi 6/7中的dbExpress组件一样。这样设计的好处是性能非常好，而缺点是这些.NET组件只能供 Borland的产品或是语言使用，因为其他语言和产品（例如 C#）不知道如何处理dbExpress接口。

所谓.NET的InterOp能力是指让.NET中的程序调用原始Windows程序的机制，例如让.NET的程序调用DLL或COM等程序和组件。请读者参考.NET相关的书籍。

而兼容访问机制则是指 Borland在dbExpress的接口之上再提供一层ADO.NET的接口，如此一来不但Galileo的组件可以通过dbExpress访问数据，而且.NET的其他语言

和产品只要支持ADO.NET接口也都可以通过这样的技术使用 dbExpress来访问更多的数据库。

图13 5说明了兼容访问机制的实现结构。

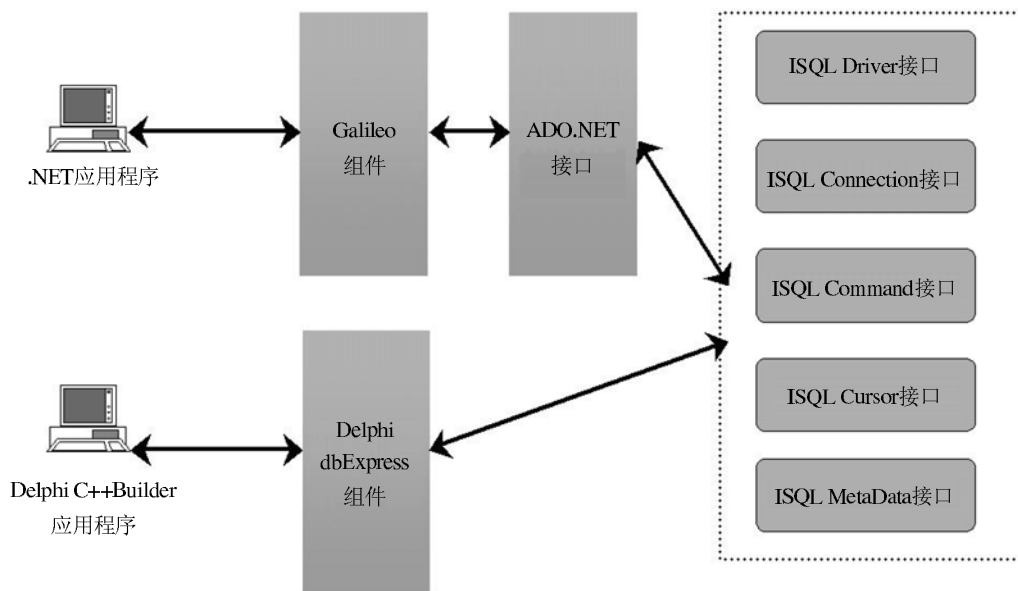


图13-5 dbExpress在未来可能的实现方式

兼容访问机制看起来是比较吸引人的方式，因为这样可以让我们 Borland 的语言和产品打入更多的潜在市场。不过也许 Borland 会同时提供这两种的实现方式，允许 Borland 的语言和产品使用较为有效率的访问方式，同时也能够让其他解决方案可以使用 dbExpress 带来的好处。

当然，在 Borland 实现 dbExpress 的兼容访问机制时也会面临许多设计和实现方面的选择和考虑因素。例如 dbExpress 接口和 ADO.NET 接口之间如何交互？dbExpress 的数据类型和 ADO.NET 数据类型如何相互转换？dbExpress 如何模拟和执行 ADO.NET 封装的嵌套数据集？这些设计和技术上的细节也是非常有趣而且值得学习的。当然，Borland 可能会发展出更为聪明的实现方式，真正的答案都会在明年的 Galileo 中揭晓。

13.5 结论

在本章中，本书从分析 dbExpress 的内容开始讨论 dbExpress 驱动程序的进入点，以及每一个 dbExpress 驱动程序如何使用不同的底层实现技术来提供连接数据库的能力。接着，本章讨论了 Borland 在设计 dbExpress 时考虑因素的技术细节，之后本章开

始说明dbExpress规定的标准接口以及每一个dbExpress接口提供的服务，最后本章也说明了dbExpress接口之间的关系。

在了解了dbExpress的运作流程以及dbExpress接口的意义之后，本章使用了一个简单的范例来说明如何实现dbExpress驱动程序，让读者更能够了解和掌握dbExpress技术。最后本章说明了Borland在未来把dbExpress移植到.NET平台时需要考虑的问题以及必须克服的技术问题，让读者了解dbExpress在未来可能的发展。

本章主要是希望读者能够对dbExpress有进一步的了解，更重要的是希望读者在使用dbExpress时也能够花一些时间思考 and 了解dbExpress是如何开发出来的。如果读者能够了解dbExpress设计的精神并且掌握dbExpress实现的技术，那么也可以在设计和开发读者自己的应用系统时使用类似的技巧。不管读者如何利用本章对dbExpress的讨论，了解更多的接口程序设计技巧都对读者提升自己的软件技术有一定的帮助，也希望读者在阅读完本章之后有所进步。

参考资料

- dbExpress文档。
- Microsoft .NET文档。
- Ramesh Theivendran所著的《Internals of dbExpress》。